TRANSMITTAL FORM

(to be used for all correspondence after initial filing)

Application Number	09/944,685
Filing Date	August 31, 2001
First Named Inventor	Henry Fang
Art Unit	2122
Examiner Name	Vo Ted T.
Total Number of Pages in This Submission	Attorney Docket Number	SLA1070

OIP FEE MAY 3 1 2005

BEST AVAILABLE COPY

ENCLOSURES *(check all that apply)*

- ☒ Fee Transmittal Form
  - ☐ Fee Attached
- ☐ Amendment / Reply
  - ☐ After Final
  - ☐ Affidavits/declaration(s)
- ☐ Extension of Time Request
- ☐ Express Abandonment Request
- ☐ Information Disclosure Statement
- ☐ Certified Copy of Priority Document(s)
- ☐ Reply to Missing Parts/ Incomplete Application
  - ☐ Reply to Missing Parts under 37 CFR1.52 or 1.53

- ☐ Drawing(s)
- ☐ Licensing-related Papers
- ☐ Petition
- ☐ Petition to Convert to a Provisional Application
- ☐ Power of Attorney, Revocation Change of Correspondence Address
- ☐ Terminal Disclaimer
- ☐ Request for Refund
- ☐ CD, Number of CD(s) _____
  - ☐ Landscape Table on CD

| Remarks |

- ☐ After Allowance Communication to TC
- ☐ Appeal Communication to Board of Appeals and Interferences
- ☒ Appeal Communication to TC (Appeal Notice, Brief, Reply Brief)
- ☐ Proprietary Information
- ☐ Status Letter
- ☐ Other Enclosure(s) *(please identify below)*:

SIGNATURE OF APPLICANT, ATTORNEY, OR AGENT

Firm	Sharp Laboratories of America, Inc.
Signature	
Printed Name	David C. Ripma, Patent Counsel
Date	May 27, 2005	Reg. No.	27,672

CERTIFICATE OF TRANSMISSION/MAILING

I hereby certify that this correspondence is being deposited with the United States Postal Service with sufficient postage as first class mail in an envelope addressed to: Mail Stop Appeal Brief, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on the date shown below.

Signature	
Typed or printed name	Tatyana Shavrikova	Date	May 27, 2005

Effective on 12/08/2004. *Fees pursuant to the Consolidated Appropriations Act, 2005 (H.R. 4818).* # FEE TRANSMITTAL # For FY 2005	**Complete if Known**	
	Application Number	09/944,685
	Filing Date	August 31, 2001
☐ Applicant claims small entity status. See 37 CFR 1.27	First Named Inventor	Henry Fang
	Examiner Name	Vo Ted T.
	Art Unit	2122
TOTAL AMOUNT OF PAYMENT | ($) 500.00	Attorney Docket No.	SLA1070

METHOD OF PAYMENT (check all that apply)

☐ Check ☐ Credit Card ☐ Money Order ☐ None ☐ Other (please identify): _____

☑ Deposit Account   Deposit Account Number: 19-1457   Deposit Account Name: Sharp Laboratories

For the above-identified deposit account, the Director is hereby authorized to: (check all that apply)

☑ Charge fee(s) indicated below   ☐ Charge fee(s) indicated below, **except for the filing fee**

☑ Charge any additional fee(s) or underpayments of fee(s)   ☑ Credit any overpayments
under 37 CFR 1.16 and 1.17

WARNING: Information on this form may become public. Credit card information should not be included on this form. Provide credit card information and authorization on PTO-2038.

FEE CALCULATION

1. BASIC FILING, SEARCH, AND EXAMINATION FEES

Application Type	FILING FEES Fee ($)	FILING FEES Small Entity Fee ($)	SEARCH FEES Fee ($)	SEARCH FEES Small Entity Fee ($)	EXAMINATION FEES Fee ($)	EXAMINATION FEES Small Entity Fee ($)	Fees Paid ($)
Utility	300	150	500	250	200	100	_____
Design	200	100	100	50	130	65	_____
Plant	200	100	300	150	160	80	_____
Reissue	300	150	500	250	600	300	_____
Provisional	200	100	0	0	0	0	_____

2. EXCESS CLAIM FEES

Fee Description	Fee ($)	Small Entity Fee ($)
Each claim over 20 (including Reissues)	50	25
Each independent claim over 3 (including Reissues)	200	100
Multiple dependent claims	360	180

Total Claims	Extra Claims	Fee ($)	Fee Paid ($)
_____ - 20 or HP =	_____	x _____	= _____

HP = highest number of total claims paid for, if greater than 20.

Indep. Claims	Extra Claims	Fee ($)	Fee Paid ($)
_____ - 3 or HP =	_____	x _____	= _____

HP = highest number of independent claims paid for, if greater than 3.

Multiple Dependent Claims

Fee ($)	Fee Paid ($)
_____	_____

3. APPLICATION SIZE FEE

If the specification and drawings exceed 100 sheets of paper (excluding electronically filed sequence or computer listings under 37 CFR 1.52(e)), the application size fee due is $250 ($125 for small entity) for each additional 50 sheets or fraction thereof. See 35 U.S.C. 41(a)(1)(G) and 37 CFR 1.16(s).

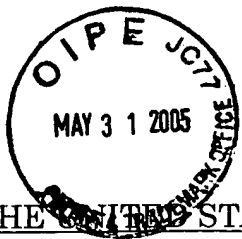| Total Sheets	Extra Sheets	Number of each additional 50 or fraction thereof	Fee ($)	Fee Paid ($)
_____ - 100 =	_____ / 50 =	_____ (round up to a whole number) x	_____	= _____

4. OTHER FEE(S)

	Fees Paid ($)
Non-English Specification, $130 fee (no small entity discount)	_____
Other (e.g., late filing surcharge): Submission of Appeal Brief	500.00

SUBMITTED BY		
Signature	Registration No. (Attorney/Agent) 27,672	Telephone 360-834-8754
Name (Print/Type) David C. Ripma, Reg. No. 27,672		Date May 27, 2005

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In Re Application of:	)	
	) ATTORNEY FILE NO.:	
Inventors: Henry Fang	) SLA1070	
	)	
Serial No.: 09/944,685	) Examiner: Vo Ted T.	
	)	
Filed: August 31, 2001	) Customer No.: 27518	
	)	
Title: SYSTEM AND METHOD FOR	) Group Art: 2122	
MANIPULATING HAVi	)	
SPECIFICATION VIRTUAL	)	
KEY DATA	) Confirmation No.: 2111	
	)	

Board of Appeals and Interferences
United States Patent and Trademark Office
P.O. Box 1450
Alexandria, VA 22313-1450
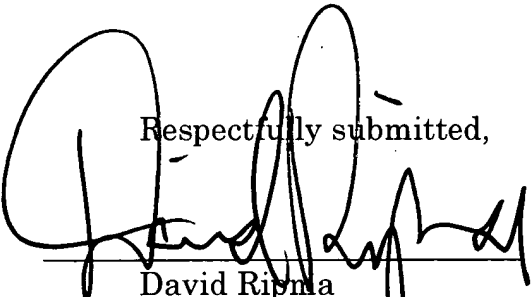
APPEAL BRIEF TRANSMITTAL LETTER

Enclosed is a copy of an Appeal Brief for the above-mentioned application, responsive to a Final Office Action mailed January 26, 2005. A Notice of Appeal for the above-mentioned application was mailed on April 26, 2005.

Please consider the enclosed Appeal Brief.

Respectfully submitted,

Date: 5/27/05

David Ripma
Registration No. 27,672

Customer Number 27518
Sharp Laboratories of America, Inc.
5750 NW Pacific Rim Blvd.
Camas, WA 98607
Telephone: (360) 834-8754
Facsimile: (360) 817-8505

BRIEF ON APPEAL

This is an appeal from the rejection by Examiner Ted T. Vo, Group Art Unit 2192, of claims 1-17 as set forth in the CLAIMS APPENDIX, all claims in the application.

REAL PARTY IN INTEREST

The real party in interest is Sharp Laboratories of America, Inc., as assignee of the present application by an Assignment in the United States Patent Office on November 13, 2001, with a recordation date of August 31, 2001 at Reel 012147, Frame 0320.

RELATED APPEALS AND INTERFERENCES

None.

STATUS OF THE CLAIMS

Claims 1-17 are in the application.

Claims 1-17 are rejected.

Claims 1-17 are appealed.

STATUS OF AMENDMENTS

Amendments were made to the claims in an Office Action response received at the PTO on September 9, 2004. The paper of September 9, 2004 was responsive to an initial Office Action mailed June 29, 2004. These claims amendments have been entered, and no further amendments have been submitted.

SUMMARY OF CLAIMED SUBJECT MATTER

The problem addressed by the present invention is presented in the specification at page 2, line 16, through page 6, line 6 (see the EVIDENCE APPENDIX, ATTACHMENT A). Generally, the problem is associated with the remote control of devices made by different manufactures, as is related to the commonly-known dilemma of a user

being forced to use a different remote controller for each piece of consumer electronics is a home entertainment system. The HAVi specification is an attempt by manufacturers to enable AV interoperability using IEEE 1394 technology. The HAVi specification has an L2 graphical user interface (GUI) application program interface (API) called HEventRepresentation that identifies the characteristics of a remote control button, which is referred to herein as a virtual key. Table 1 cross-references HAVi suggested events (functions) with symbols. Like many specifications, HAVi leaves specific implementations undefined. For example, HAVi does not define how the key information, the information shown in Table 1, is stored, or where the information is to be stored.

The invention of claim 1 is described in the specification at page 11, line 13, through page 13, line 2, and shown in Fig. 1 (see EVIDENCE APPENDIX, ATTACHMENT B). The invention describes accessing a Java ARchive (JAR) file to retrieve virtual key information. As noted on page 16, line 25, through page 17, line 7, this Java Virtual Machine (JVM) accessing model permits the virtual key information to be embedded and retrieved from a data array or static class, see Fig. 7. The JVM access method is the compromise choice when considering memory usage and programming costs (as compared to the methods of claims 7 and 12).

The invention of claim 7 is described in the specification at page 13, line 3-26, see Fig. 2. The invention retrieves virtual key information from a Java I/O ResourceBundle. As noted on page 17, line 8-23, the ResourceBundle class is used to store virtual key information as a property file. This I/O accessing model permits the virtual key information to be stored in a storage medium such as Flash memory. The

advantage of this model is that the key information can be programmed by the user, see Fig. 8.

The invention of claim 12 is described at page 14, lines 1-26, see Fig. 3. The claim recites calling a Java native interface (JNI), and using the JNI to call a mapped memory, which is accessed to retrieve virtual key information. As explained on page 17, line 24, through page 18, line 6, the JNI storage driver uses the minimum amount of memory.

GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

1. Whether claims 1-17 are indefinite under 35 U.S.C. 112, second paragraph, for failing to particularly point out and distinctly claim the subject matter.

2. Whether claims 1-17 are anticipated under 35 U.S.C. 102(a) with respect to the HAVi Specification, Version 1.1, May 15, 2001 ("HAVi").

ARGUMENT

1. The Rejection of claims 1-17 under 35 U.S.C. 112, second paragraph, as indefinite for failing to particularly point out and distinctly claim the subject matter.

Section 4 of the Final Office Action states that "(c)laims 1, 7, and 12 which recite newly limitation "defining device user interface control" in response to the prior rejection do not make the scopes of the Claims clear." More specifically, the Office Action states that "(t)he recitation in these Claims' preambles is not clear in combining with the step's functionality in the claim body, *"retrieving virtual key information"*.

There are lacking essential elements or features in the claim to connect the preamble, particularly the preamble recitation "*defining device user interface controls*"."

The second paragraph of 35 U.S.C. 112 states that "(t)he patent specification shall conclude with one or more claims particularly pointing out and distinctly claiming the subject matter of the invention." As noted in MPEP 2171, the second paragraph of U.S.C. 112, recites two requirements; that the claim sets forth the subject matter of the invention, and that the claims set forth the metes and bounds of the subject matter.

As noted in MPEP 2173.02, "(i)n reviewing a claim for compliance with 35 U.S.C. 112, second paragraph, the examiner must consider the claim as a whole to determine whether the claim apprises one of ordinary skill in the art of its scope, and therefore, serves the notice function required by 35 U.S.C. 112, second paragraph..." *Solomon v. Kimberly-Clark Corp.*, 216 F.3d 1372, 1379, 55 USPQ2d 1279, 1283 (Fed. Cir. 2000).

Even more to the point, MPEP 2173.05(e) states that "(t)he mere fact that the body of the claim recites additional elements which do not appear in the claim's preamble does not render the claim indefinite under 35 U.S.C. 112, second paragraph *In re Larson*, No. 01-1092 (Fed. Cir. May 9, 2001).

The preambles of claims 1, 7, and 12, recite a method for defining device user interface control. Each independent claim recites a different method for accessing the virtual key information, which defines the device user interface.

In summary, the Applicant respectfully submits that there is no requirement that the preamble of a claim list every claim element recited in the body of the claim. Further, the Applicant submits that a claim cannot be judged as indefinite merely for what is recited, or not recited in the preamble. Finally, the Applicant submits that, read as a whole, claims 1, 7, and 12 definitely claim the subject matter of an invention that retrieves virtual key information as a specific method of defining device user interface controls.

2. *The Rejection of claims 1-17 under 35 U.S.C. 102(a) as anticipated by the HAVi Specification, Version 1.1, May 15, 2001 ("HAVi").*

Section 7 of the Final Office Action states that claims 1-17 have been rejected under 35 U.S.C. 102(a) as being anticipated by HAVi. With respect to claim 1, the Office Action states that HAVi discloses the retrieval of virtual key information in response to accessing a JAR file (Section 1.3, page 5, 2.5.2, and 2.7.2). Sections 2.9.2 and Section 8.3.2.5 (page 429) are referenced with respect to retrieving virtual key information.

With respect to claim 7, the Office Action states that HAVi discloses the retrieval of virtual key information in response to accessing a ResourceBundle. The Office Action references Sections 1.3, 2.7.2, 2.5.2, 2.9.2, 8.3.2.4, and 8.3.2.5. With respect to claim 12, the Office Action states that HAVi discloses the retrieval of virtual key information in response to accessing a mapped memory, referencing Sections 1.3, 2.7.2, 2.5.2, 2.9.2, 7.4, 8.1, 8.7, 8.8, 3.10.1, 9.4, 9.5, and 8.3.2.5.

The *Response to Arguments* Section of the Office Action states (page 3, second-last paragraph) that, "...the above Applicant's assertions tend to short, not to extend to the whole HAVi specification which incorporates the Claim's limitations. Respectfully, the whole consideration of Havi Specification would be required."

The Applicant assumes that the above-quoted section of the Office Action is intended to mean that the Applicant's claims must be read in the context of the entire HAVi specification, not just the specific HAVi specification sections listed in the Office Action. However, the Applicant is unaware of any other sections, or combinations in the HAVi specification that can be used to support the Examiner's assertions. Further, the Applicant notes that it is the Examiner's responsibility to establish a *prima facie* case to support a rejection. As described in detail below, the portions of HAVi that have been cited in the Office Action do not anticipate the claimed invention.

Claims 1 through 6

The Final Office Action cites the following HAVi Sections as anticipating claim 1:

Section 1.3 is a chart of terminology, which at page 5, line 10, lists, "HAVi Level 2 interoperability" (see EVIDENCE APPENDIX, ATTACHMENT C). An expert in the art may use the information in this section to enable a compatible implementation for supporting code upgrades. The Applicant notes that this Section does not state how an implementation is to be achieved.

Section 2.5.2 states that the L2 UI is based upon JAVA AWT 1.1. An expert in the art may use the information in this section to

understand the required scope and restricted support of JAVA AWT 1.1. APIs, employed to present UI features of interests.

Section 2.7.2 describes Level 2 interoperability. An expert in the art may use the information in this section to understand the Level 2 Interoperability specification with respect to: (1) interoperability among devices, and (2) the variety of design "flavors" that may be used by manufacturers within the scope of the specification.

Section 2.9.2 describes Signature Verification. An expert in the art may use the information in this section to understand the signature verification mechanism recommended to maintain data integrity and security.

Section 8.3.2.5 (page 429) states that an event can have a representation as a string, color, or symbol, which can be determined by calling *get*String, *get*Color, and *get*Symbol, respectively. This methodology permits the definition of a device button. An expert in the art may use the information in this section to insure that their design implementations recognize specified and enumerated keycodes. These *get*Sting(), *get*Color, and *get*Symbol() methods direct a HAVi device to get the color, prompt sting, and icon information to be displayed. However, this Section does not specifically describe the controls to be used to retrieve the information to be seen in the UI. That is, the specification is abstract enough to leave the specific retrieval mechanism to the discretion of the various manufacturers. The claimed invention may be understood as a specific means that facilitates easy upgrades, modularized for customize orders and other business advantages, which can be inexpensively performed in non-engineering and non-manufacturing environments.

None of the above-mentioned HAVi Sections teach how the virtual key information is to be stored or retrieved from memory. In contrast, claim 1 recites the retrieval of virtual key information in response to accessing a JAR file stored in memory.

With respect to claim 1, the ***Response to Arguments*** Section (last full paragraph of page 4) states that Section 2.5.2, 7.2.2, and 3.10.1 disclose the accessing of virtual key information from JAR files. In response, the Applicant notes that the cited portions of the HAVi specification do not describe the retrieval of virtual key information, in response to accessing a JAR file, as recited in claim 1.

Section 2.5.2, summarized above, describes a Level 2 user interface that can be used to support display screen functions, alpha blending, remote control inputs, and support for visual interface components. Nowhere in this section is there a description of accessing a JAR file to retrieve virtual key information.

Section 7.2.2, at page 394, describes the packages and classes that may be used in DCM and Application Module code units. An expert in the art may use the information in this section to understand the full AV device (FAV) class of HAVi devices, which must accommodate Havlets, and the Section 2.5.2 implementation that should reside in this class. Nowhere in this section is there a description of accessing a JAR file to retrieve virtual key information.

Fig. 25 (Section 3.10.1), at page 89, describes the retrieval and use of digital signatures. An expert in the art may use the information in this section to generally implement JAR file security mechanisms. This Section does not discuss the downloading of UI

contents. Section 3.10.1 does not include a description of accessing a JAR file to retrieve virtual key information.

In summary, the cited HAVi specifications do not describe accessing a JAR file to retrieve virtual key information. These cited sections, and other uncited sections of the HAVi specification are general implementation guidelines. Claim 1, while operating in the context of the HAVi specification, is a narrower invention that includes limitations that cannot be found in the HAVi specification.

"A claim is anticipated only if each and every element as set forth in the claim is found, either expressly or inherently described, in a single prior art reference." *Verdegaal Bros. v. Union Oil of California*, 814 F.2d 628, 631, 2 USPQ2d 1051, 1053 (Fed. Cir. 1987).

In short, HAVi does not give any guidance as to the virtual key information storage format. Claim 1 describes the virtual key information stored in a JAR file format. The HAVi specification does not explicitly describe every element of the claimed invention. Since HAVi does not describe every limitation, it cannot anticipate. Claims 2-6, dependent from claim 1, enjoy the same distinctions from the cited prior art.

Claims 7 through 11

With respect to claim 7, the cited HAVi specification Sections include:

Section 1.3, as described above, is a chart of terminology, which at page 5, line 10, lists, "HAVi Level 2 interoperability".

Section 2.5.2, as described above, states that the L2 UI is based upon JAVA AWT 1.1.

Section 2.7.2, as described above, describes Level 2 interoperability.

Section 2.9.2, as described above, describes Signature Verification.

Section 8.3.2.4 states that there are three classes available to determine if a device is available. An expert in the art may use the information in this section to implement support for color fetching, prompt fetching, and icon fetching. Without the supported implementations, color, prompt, and icon information cannot be accessed.

Section 8.3.2.5 (page 429), as described above, states that an event can have a representation as a string, color, or symbol, which can be determined by calling *get*String, *get*Color, and *get*Symbol, respectively.

Claim 7 recites the retrieval of virtual key information in response to accessing a ResourceBundle. None of the above-mentioned HAVi Sections teach the retrieval of virtual key information in response to accessing a ResourceBundle.

The *Response to Arguments* Section of the Office Action states that that Section 8.3.2.5 of the HAVi specification describes input elements such as String, Color, and Symbol. The *Response to Arguments* Section continues, stating that 8.3.2.5 and Section 2.9.2 describe 6 keys that may be obtained from calling "getColor".

In response, the Applicant notes that Section 8.3.2.4 states that there are three classes available to determine if a device is available. Section 8.3.2.5 states that an event (device button) can have a representation such as a string, color; or symbol, which can be determined by calling *get*String, *get*Color, and *get*Symbol, respectively. However, nowhere in these sections is there a description of accessing a

ResourceBundle to retrieve virtual key information. Alternately stated, HAVi specifies_getString, getColor, and getSymbol functions as a programming interface to users. However, the underlining implementation is transparent to users. A ResourceBundle may be employed in the claimed invention to support international character sets, for example, for simple configuration in a non-manufacturing environment.

Section 2.9.2 of the HAVi specification describes a signature verification process. Nowhere in this section is there a description of accessing a ResourceBundle to retrieve virtual key information.

Claim 7, while operating in the context of the HAVi specification, is a narrower invention that includes limitations that cannot be found in the HAVi specification. Claim 7 describes the virtual key information stored in a ResourceBundle. The HAVi specification does not explicitly describe every element of the claimed invention. Since HAVi does not describe every limitation, it cannot anticipate. Claims 8-11, dependent from claim 7, enjoy the same distinctions from the cited prior art.

Claims 12 through 17

With respect to claim 12, the Final Office Action cites the following HAVi Specification Sections:

Section 1.3, as described above, is a chart of terminology, which at page 5, line 10, lists, "HAVi Level 2 interoperability".

Section 2.5.2, as described above, states that the L2 UI is based upon JAVA AWT 1.1.

Section 2.7.2, as explained above, describes Level 2 interoperability.

Section 2.9.2, as explained above, describes Signature Verification.

Section 7.4 states that JAVA code units are entities for uploading, and that the format of a JAVA code unit is the JAR format. Details are given of DCM, AMC, and Havlet code units. An expert in the art may use this section to implement downloadable contents, and a Havlet as a delivering software entity.

Section 8.1 describes the HAVi user interface design, using a subset of AWT as defined in JAVA 1.1. An expert in the art may use this section to determine that the API set is limited to JAVA AWT 1.1. No other "luxurious" support can be expected.

Section 8.7 describes a general approach to error behavior. An expert in the art may use this section to implement error-handling mechanism when fault conditions arise.

Section 8.8 is a list of constants. An expert in the art may use this section to implement a device with those specified enumerates, so as to be compatible to other devices using the same specification.

Fig. 25 (Section 3.10.1), at page 89, as explained above, describes the retrieval and use of digital signatures.

Section 9.4 presents a list of defined HAVi key values. An expert in the art may use this section to implement compatible devices, which understand the commands originating from external devices.

Section 9.5 lists HAVi and non-HAVi ROM requirements. An expert in the art may use this section to implement HAVi code together with other system code using IEC-61883 format.

Section 8.3.2.5 (page 429), as described above, states that an event can have a representation as a string, color, or symbol, which can be determined by calling *get*String, *get*Color, and *get*Symbol, respectively.

Claim 12 recites the retrieval of virtual key information in response using a JNI to access a mapped memory. None of the above-mentioned HAVi Sections teach the use of a JNI to access mapped memory, to retrieve virtual key information.

The ***Response to Arguments*** Section of the Office Action states that the Level 2 UI provides access to keys to a code like a JAR file that resides in memory (Section 7.4). The Office Action states that the format of a Java Code unit is the Java archive or JAR format. The Office Action states that page 89 shows HAVi uploading/obtaining keys, that Sections 9.4 and 9.5 show key representations, and that Section 9.7 discusses means for accessing mapped memory where JAR native code resides.

In response, the Applicant notes that Section 7.4 of the HAVi specification describes that JAVA code units are entities for uploading, and that the format of a JAVA code unit is the JAR format. Nowhere in this section is there a description of the retrieval of virtual key information in response using a JNI to access a mapped memory.

Fig. 25 (Section 3.10.1), at page 89, describes the retrieval and use of digital signatures. Section 3.10.1 does not include a description of the retrieval of virtual key information in response using a JNI to access a mapped memory. Section 9.4 presents a list of defined HAVi key values. Section 9.5 lists HAVi and non-HAVi ROM requirements. Neither of these sections describes the retrieval of virtual key information in response using a JNI to access a mapped memory. Section 9.7 of the

HAVi specification describes the GUID and Bus_Info_Block. This section does not describe the retrieval of virtual key information in response using a JNI to access a mapped memory.

Claim 12 describes the virtual key information stored as mapped memory accessible using JNI. The HAVi specification does not explicitly describe every element of the claimed invention. Since HAVi does not describe every limitation, it cannot anticipate. Claims 13-17, dependent from claim 12, enjoy the same distinctions from the cited prior art.

In summary, the HAVi specification does not explicitly describe claims 1, 7, and 12, because the HAVi specification does not describe any mechanisms for actually implementing a class of virtual key representations. The specification only offers general guidelines. That is, the HAVi specification does not describe how virtual key information is to be stored, or where it is to be stored. Therefore, with respect to claim 1, none of the HAVi sections cited in the Office Action, whether considered independently, or as a group, describe the step of accessing a JAR file to retrieve virtual key information. With respect to claim 7, none of the cited sections describe a method of accessing a ResourceBundle to retrieve of virtual key information. With respect to claim 12, none of the cited sections describe the retrieval of virtual key information in response using a JNI to access a mapped memory. The cited references do not explicitly describe every limitation of claims 1, 7, and 12. As a matter of well-established law, a claim cannot be anticipated if the prior art reference does not explicitly describe every limitation of the claimed invention. Since the HAVi specification does not explicitly describe every limitation of claims 1, 7, and 12, it cannot anticipate. Claims 2-6, dependent from

claim 1, claims 8-11, dependent from claim 7, and claims 13-17, dependent from claim 12, enjoy the same benefits.

SUMMARY AND CONCLUSION
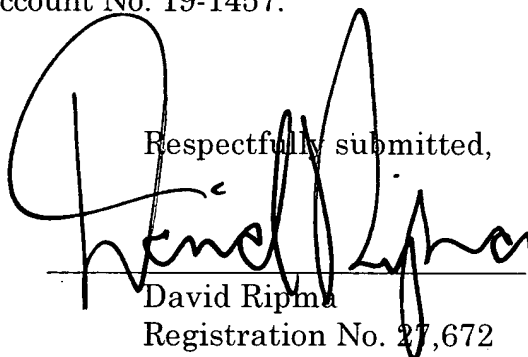
It is submitted that for the reasons pointed out above, the claims in the present application clearly and patentably distinguish over the cited references. Accordingly, the Examiner should be reversed and ordered to pass the case to issue.

Authorization is provide, in the amount of $500.00, to cover the fee for this Appeal Brief. Authorization is given to charge any deficit or credit any excess to Deposit Account No. 19-1457.

Respectfully submitted,

Date: 5/27/05

David Ripma
Registration No. 27,672

Customer Number 27518
David Ripma, Patent Counsel
Sharp Laboratories of America, Inc.
5750 NW Pacific Rim Blvd.
Camas, WA 98607
Telephone: (360) 834-8754
Facsimile: (360) 817-8505
dripma@sharplabs.com

TABLE OF CONTENTS

ATTACHMENT A (Applicant's Specification)

ATTACHMENT B (Applicant's Drawings)

ATTACHMENT C (The HAVi Specification, Version
1.1, May 15, 2001)

CLAIMS APPENDIX

1.   (Previously Presented)   In a signal bearing medium tangibly embodying a program of machine-readable instructions executed by an open source, interoperable IEEE 1394 specification digital device, a method for defining device user interface controls, the method comprising:

from a level two (L2) graphical user interface (GUI), accessing a JAR file; and,

in response to accessing the JAR file, retrieving virtual key information.

2.   (Original)   The method of claim 1 wherein accessing a JAR file includes accessing a JAR file stored in read only memory (ROM).

3.   (Original)   The method of claim 2 wherein retrieving virtual key information includes retrieving virtual key information from a JAR file model selected from the group including static classes and data arrays.

4.   (Previously Presented)   The method of claim 3 wherein retrieving virtual key information in response to accessing the JAR file includes retrieving an application bundled with the virtual key information.

5.   (Original)   The method of claim 4 in which a first microprocessor machine using a first operating system is included;

the method further comprising:

receiving virtual key information as Java source code;

using a Java compiler, compiling the Java source code into Java virtual machine (JVM) byte codes for the first operating system; and,

using jar tools, archiving the JVM byte codes into a JAR file stored in ROM.

6.   (Previously Presented)   The method of claim 5 further comprising:

receiving the application as Java source code;

using a Java compiler, compiling the Java source code into Java virtual machine (JVM) byte codes for the first operating system; and,

using jar tools, archiving the JVM byte codes into a JAR file stored in ROM.

7.   (Previously Presented)   In a signal bearing medium tangibly embodying a program of machine-readable instructions executed by an open source, interoperable IEEE 1394 specification digital device, a method for defining device user interface controls, the method comprising:

from a level two (L2) graphical user interface (GUI) accessing a Java input/output (I/O) ResourceBundle; and,

in response to accessing the ResourceBundle, retrieving virtual key information.

8.   (Original)   The method of claim 7 wherein accessing the ResourceBundle includes using a ResourceBundle application program interface (API) to specify a property file.

9.   (Previously Presented)   The method of claim 8 in which a first microprocessor machine using a first operating system is included;

the method further comprising:

maintaining an application in a protocol associated with the first operating system; and,

wherein accessing the ResourceBundle includes using a ResourceBundle API to specify a property file stored in a file system associated with the first microprocessor machine.

10.   (Original)   The method of claim 9 wherein using a ResourceBundle API to specify a property file stored in the file system includes specifying a property file stored in an input/output (I/O) device selected from the group of storage devices including hard disks and Flash memory.

11.   (Original)   The method of claim 10 further comprising:

receiving virtual key information as text-based properties attributes in a ResourceBundle property file;

integrating the virtual key information into a table of virtual key characteristics; and,

storing the virtual key characteristics table as the ResourceBundle property file.

12.   (Previously Presented)   In a signal bearing medium tangibly embodying a program of machine-readable instructions executed by an open source, interoperable IEEE 1394 specification digital device, a method for defining device user interface controls, the method comprising:

from a level two (L2) graphical user interface (GUI) calling a Java native interface (JNI);

at the JNI, using Java byte codes to call a storage driver;

from the storage driver, accessing a mapped memory; and,

in response to accessing the mapped memory, retrieving virtual key information.

13.   (Original)   The method of claim 12 wherein accessing a mapped memory includes accessing a mapped memory stored in an electrically erasable programmable read only memory (EEPROM).

14.   (Original)   The method of claim 13 wherein retrieving virtual key information includes retrieving virtual key information from mapped memory in a binary format.

15.   (Original)   The method of claim 14 wherein using Java byte codes to call a storage driver at the JNI includes converting the Java byte code to binary format addresses; and,

wherein accessing a mapped memory from the storage driver includes using the binary format addresses to access ASCII codes stored in the EEPROM.

16. (Original) The method of claim 15 in which a first microprocessor using a first operating system is included;

the method further comprising:

receiving the storage driver as first operating system machine codes; and,

storing the storage driver as machine code.

17. (Original) The method of claim 16 further comprising:

receiving virtual key information as binary format code;

using the storage driver, cross-referencing the virtual key information with EEPROM addresses; and,

storing the virtual key information in the EEPROM as machine code.

EVIDENCE APPENDIX

ATTACHMENT A

SYSTEM AND METHOD
FOR MANIPULATING HAVi
SPECIFICATION VIRTUAL KEY DATA

BACKGROUND OF THE INVENTION

1.   Field of the Invention

This invention generally relates to home audiovisual systems and, more particularly, to a method for storing and accessing HAVi compliant virtual key event representation information.

2.   Description of the Related Art

As noted in US Patent 6,032,202 (Lea), a typical home audiovisual equipment set up includes a number of components. For example, a radio receiver, a CD player, a pair of speakers, a television, a VCR, a tape deck, and alike. Each of these components is connected to each other via a set of wires. One component is usually the central component of the home audiovisual system. This is usually the radio receiver, or the tuner. The tuner has a number of specific inputs for coupling the other components. The tuner has a corresponding number of control buttons or control switches that provide a limited degree of controllability and interoperability for the components. The control buttons and control switches are usually located on the front of the tuner. In many cases, some, or all, of these buttons and switches are duplicated on a hand held remote control unit. A user controls the home audiovisual system by manipulating the buttons and switches on the front of the tuner, or alternatively, manipulating buttons on the hand held remote control unit.

This conventional home audiovisual system paradigm has become quite popular. As consumer electronic devices become more

capable and more complex, the demand for the latest and most capable devices has increased. As new devices emerge and become popular, the devices are purchased by consumers and "plugged" into their home audiovisual systems. Generally, the latest and most sophisticated of these

5   devices are quite expensive (e.g., digital audio tape recorders, DVD players, digital camcorders, and alike). As a consumer purchases new devices, most often, the new device is simply plugged into the system alongside the pre-existing, older devices (e.g., cassette tape deck, CD player, and the like). The new device is plugged into an open input on the

10   back of the tuner, or some other device couple to the tuner. The consumer (e.g., the user) controls the new device via the control buttons on the tuner, via the control buttons and control switches on the front of the new device itself, or via an entirely new, separate, respective remote control unit for the new device.

15   As the number of new consumer electronics devices for the home audiovisual system have grown and as the sophistication and capabilities of these devices have increased, a number of problems with the conventional paradigm have emerged. One such problem is incompatibility between devices in the home audiovisual system.

20   Consumer electronic devices from one manufacturer often couple to an audiovisual system in a different manner than similar devices from another manufacturer. For example, a tuner made by one manufacturer may not properly couple with a television made by another manufacturer.

In addition, if one device is much newer than another device,

25   additional incompatibilities may exist. For example, a new device might incorporate hardware (e.g., specific inputs and outputs) that enables more

sophisticated remote control functions. This hardware may be unusable with older devices within the system. Or, for example, older tuners may lack suitable inputs for some newer devices (e.g., mini-disc players, VCRs, etc.), or may lack enough inputs for all devices of the system.

5   Another problem is the lack of functional support for differing devices within an audiovisual system. For example, even though a television may support advanced sound formats (e.g., surround sound, stereo, etc.), if an older less capable tuner does not support such functionality, the benefits of the advanced sound formats can be lost.

10   Another problem is the proliferation of controls for the new and differing devices within the home audiovisual system. For example, similar devices from different manufacturers can each have different control buttons and control switch formats for accomplishing similar tasks (e.g., setting the clock on a VCR, programming a VCR record a later

15   program, and alike). In addition, each new device coupled to the audiovisual system often leads to another dedicated remote control unit for the user to keep track of and learn to operate.

        The emergence of networking and interface technology (e.g., IEEE 1394 serial communication bus and the wide spread adoption of

20   digital systems) offers prospects for correcting these problems. Further, an alliance of manufactures has agreed to an open, extensible architecture to provide for intelligent, self configuring, easily extensible devices or AV systems.

        It should be noted that the home AV interoperability (HAVi)

25   architecture of the present invention is an open, platform-independent, architecturally-neutral network that allows consumer electronics

manufacturers and producers to provide inter-operable appliances. It can be implemented on different hardware/software platforms and does not include features that are unique to any one platform. The interoperability interfaces of the HAVi architecture are extensible and can be added to,

5 modified, and advanced as market requirements and technology change. They provide the infrastructure to control the routing and processing of isochronous and time-sensitive data (e.g., such as audio and video content).

Specifically, the HAVi architecture provides: an execution

10 environment supporting the visual representation and control of appliances; application and system services; and communication mechanisms for extending the environment dynamically through plug and play or otherwise.

The underlying structure for such a network consists of set of

15 interconnected clusters of appliances. Typically, there will be several clusters in the home, with one per floor, or per room. Each cluster will work as a set of interconnected devices to provide a set of services to users. Often, one device will act as a controller for a set of other devices. However, the architecture is sufficiently flexible to also allow a home to

20 consist of a single cluster with no master controller.

The level two (L2) graphical user interface (GUI) of HAVi system includes an application program interface (API) called HEventRepresentation to identify the characteristics of remote control, or front panel buttons. A conventional implementation of the

25 HEventRepresentation class API consists of specification compliant portions, that permit interoperation between different devices and

manufacturers. However, implementation specific parts of the API can vary for different manufacturers, to highlight certain device characteristics, for example, or for use in creating features by the system user.

5   In the HAVi system, all the applications that reference the HEventRepresentation API request information concerning a display key or button specified in HAVi specification. These display keys or buttons are referred to herein as virtual keys or keys, since the HEventRepresentation API is typically invoked to provide information, so

10   that a button can be displayed on a TV screen or liquid crystal display (LCD) panel. The HAVi specification suggests that the HEventRepresentation API provide information for the keys shown in Table 1.

       The HAVi specification suggests that applications use the

15   HEventRepresentation.getString(), the HEventRepresentation.getColor(), and the HEventRepresentation.getSymbol() to retrieve information about each event (key) represented in the HAVi system.

Event	Implied symbol	Sample
VK_GO_TO_START	Two equilateral triangles, pointing at a line to the left	⏮
VK_REWIND	Two equilateral triangles, pointing to the left	⏪
VK_STOP	A square	⏹
VK_PAUSE	Two vertical lines, side by side	⏸
VK_PLAY	One equilateral triangle, pointing to the right	▶
VK_FAST_FWD	Two equilateral triangles, pointing to the right	⏩
VK_GO_TO_END	Two equilateral triangles, pointing to a line at the right	⏭
VK_TRACK_PREV	One equilateral triangle, pointing to a line at the left	⏮
VK_TRACK_NEXT	One equilateral triangle, pointing to a line at the right	⏭
VK_RECORD	A circle, normally red	●
VK_EJECT_TOGGLE	A line under a wide triangle which points up	⏏
VK_VOLUME_UP	A ramp, increasing to the right, near a plus sign	◿
VK_VOLUME_DOWN	A ramp, increasing to the right, near a minus sign	◿
VK_UP	An arrow pointing up	⇧
VK_DOWN	An arrow pointing down	⇩
VK_LEFT	An arrow pointing to the left	⇦
VK_RIGHT	An arrow pointing to the right	⇨
VK_POWER	A circle, broken at the top, with a vertical line in the break	⏻

Table 1: Sample of HEventRepresentation class contents.

However, the HAVi specification does not suggest how the
key information, shown in Table 1, should be stored. Nor does the
specification specify where the information should be stored.

It would be advantageous if a HAVi compliant
HEventRepresentation method could be found that permitted the features
of particular devices to be highlighted using virtual keys.

It would be advantageous if a HAVi compliant
HEventRepresentation method could be found that minimized
maintenance costs associated with identifying the table contents to be

modified, creating code to use the table, modifying the table contents, and verifying the table content modifications.

It would be advantageous if a HAVi compliant HEventRepresentation method could be found that minimized the amount

5   of memory required to store virtual key information.

It would be advantageous if a HAVi compliant HEventRepresentation method could be found that permitted a designer to make trade-offs between programmability, memory, and maintenance costs.

10

SUMMARY OF THE INVENTION

The HAVi specification offers a guideline as to how a class of virtual key representations can be implemented for applications that display virtual keys. The HAVi specification is an implementation

15   guideline, but the actual implementation is dependent upon the platform; the microprocessor system and the software operating system. The actual implementation is also dependent upon specific device features and resource requirements.

Accordingly, a method is provided for the maintaining

20   HEventRepresentation virtual keys in a device using HAVi specification protocols. The method comprises: from a HAVi level two (L2) graphical user interface (GUI), accessing a Java ARchive (JAR) file; and, in response to accessing the JAR file from read only memory (ROM), retrieving virtual key information as a static class or data array.

25   Alternately, the method comprises: from a HAVi L2 GUI accessing a Java input/output (I/O) ResourceBundle; and, in response to

accessing the ResourceBundle, retrieving virtual key information stored in an input/output (I/O) device such as a hard disk or Flash memory.

In yet another alternative, the method comprises: from a HAVi L2 GUI calling a Java native interface (JNI); at the JNI, using Java

5   byte codes to call a storage driver; from the storage driver, accessing a mapped memory stored in binary format in an electrically erasable programmable read only memory (EEPROM); and, in response to accessing the mapped memory, retrieving virtual key information.

Additional details of the maintaining HEventRepresentation

10   virtual keys are provided below.

BRIEF DESCRIPTION OF THE DRAWING

Fig. 1 is a flowchart illustrating a first method for maintaining HEventRepresentation virtual keys in a device using HAVi

15   specification protocols.

Fig. 2 is a flowchart illustrating a second method for maintaining HEventRepresentation virtual keys in a device using HAVi specification protocols.

Fig. 3 is a flowchart illustrating a third method for

20   maintaining HEventRepresentation virtual keys in a device using HAVi specification protocols.

Fig. 4a is an example property text file virtual key representation.

Fig. 4b depicts the process of using a ResourceBundle to

25   access virtual key information.

Fig. 5 illustrates an example text array virtual key representation.

Fig. 6 is an example of a static class virtual key representation.

5   Fig. 7 is a representation of a JVM accessing model.

Fig. 8 is a representation of a Java I/O accessing model.

Fig. 9 is a representation of a JNI/storage driver accessing model.

Fig. 10 is an example display of virtual keys created with

10   string data and color data.

Fig. 11 is an example display of virtual keys using symbols data.

DETAILED DESCRIPTION OF THE PREFERRED

15   EMBODIMENTS

Some portions of the detailed descriptions that follow are presented in terms of procedures, steps, logic blocks, codes, processing, and other symbolic representations of operations on data bits within a microprocessor or memory. These descriptions and representations are

20   the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. A procedure, microprocessor executed step, application, logic block, process, etc., is here, and generally, conceived to be a self-consistent sequence of steps or instructions leading to a desired result. The steps are

25   those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or

magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated in a microprocessor device. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms,

5 numbers, or the like. Where physical devices, such as a memory are mentioned, they are connected to other physical devices through a bus or other electrical connection. These physical devices can be considered to interact with logical processes or applications and, therefore, are "connected" to logical operations. For example, a memory can store or

10 access code to further a logical operation, or an application can call a code section from memory for execution.

It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

15 Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as "processing" or "connecting" or "translating" or "displaying" or "prompting" or "determining" or "displaying" or "recognizing" or the like, refer to the action and processes

20 of in a microprocessor system that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the wireless device memories or registers or other such information storage, transmission or display devices.

25 The present invention is based on HAVi specification V1.1, and, more particularly, the Level 2 User Interface (Chapter 8), which is

incorporated herein by reference.   The present invention can generally be described as follows:

(1)   three data formats for storing event representation (virtual key representation) in the HAVi L2 system;

5   (2)   three media models (devices) to contain data formats described in (1) above; and,

(3)   user programmable event representation.

Fig. 1 is a flowchart illustrating a first method for maintaining HEventRepresentation virtual keys in a device using HAVi

10   specification protocols. The maintenance method concerns virtual key information storage and access to the stored information. Although the method, and the methods described below, has been depicted as a sequence of steps for clarity, no order should be inferred from the numbering unless explicitly stated. The method begins at Step 100. Step

15   102, from a HAVi level two (L2) graphical user interface (GUI), accesses a JAR file. Step 104, in response to accessing the JAR file, retrieves virtual key information.

As is well known, a JAR is a file format used to include all the elements required by a Java application. Downloading is simplified

20   since the files needed to support the "applet" are bundled together with the "applet" (strictly speaking an applet is executed in a browser environment). In the present invention the "applet" can be thought of as the HEventRepresentation application. Once the file is identified, it is downloaded and separated into its components. During the execution of

25   the "applet", when a new class or data array is requested by the "applet", it is searched for (first) in the archives associated with the "applet".

Accessing a JAR file in Step 102 includes accessing a JAR file stored in read only memory (ROM). Retrieving virtual key information in Step 104 includes retrieving virtual key information from a JAR file model selected from the group including static classes and data arrays.

5   Retrieving virtual key information in response to accessing the JAR file in Step 104 also includes retrieving a HEventRepresentation application bundled with the virtual key information.

In some aspects of the invention, a first microprocessor machine using a first operating system is included. Then, the method

10   comprises further steps. Prior to accessing the JAR file in Step 102, the method comprises further steps. Step 101a1 receives virtual key (VK) information as Java source code. Step 101b1, using a Java compiler, compiles the Java source code into Java virtual machine (JVM) byte codes for the first operating system. Step 101c1, using jar tools, archives the

15   JVM byte codes into a JAR file stored in ROM.

Step 101a2 receives the HEventRepresentation application as Java source code. Step 101b2, using a Java compiler, compiles the Java source code into Java virtual machine (JVM) byte codes for the first operating system. Step 101c2, using jar tools, archives the JVM byte

20   codes into a JAR file stored in ROM.

Fig. 2 is a flowchart illustrating a second method for maintaining HEventRepresentation virtual keys in a device using HAVi specification protocols. The method starts at Step 200. Step 202, from a HAVi L2 GUI, accesses a Java input/output (I/O) ResourceBundle. Step

25   204, in response to accessing the ResourceBundle, retrieves virtual key

information. Accessing the ResourceBundle in Step 204 includes using a ResourceBundle API to specify a property file.

A first microprocessor machine using a first operating system is included. Then, the method comprises a further step. Step 201a

5   maintains a HEventRepresentation application in a protocol associated with the first operating system. Accessing the ResourceBundle in Step 204 includes using a ResourceBundle API to specify a property file stored in a file system associated with the first microprocessor machine. Further, using a ResourceBundle API to specify a property file stored in

10   the file system in Step 204 includes specifying a property file stored in an I/O device selected from the group of storage devices including hard disks and Flash memory.

The method comprises further steps. Step 201b receives virtual key information as text-based properties attributes in a

15   ResourceBundle property file. Step 201c integrates the virtual key information into a table of virtual key characteristics. Step 201d stores the virtual key characteristics table as the ResourceBundle property file.

Fig. 3 is a flowchart illustrating a third method for maintaining HEventRepresentation virtual keys in a device using HAVi

20   specification protocols. The method begins at Step 300. Step 302, from a HAVi L2 GUI, calls a Java native interface (JNI). Step 304, at the JNI, uses Java byte codes to call a storage driver. Step 306, from the storage driver, accesses a mapped memory. Step 308, in response to accessing the mapped memory, retrieves virtual key information.

25   Accessing a mapped memory in Step 306 includes accessing a mapped memory stored in an electrically erasable programmable read

only memory (EEPROM). Retrieving virtual key information in Step 308 includes retrieving virtual key information from mapped memory in a binary format.

Using Java byte codes to call a storage driver at the JNI in

5   Step 304 includes converting the Java byte code to binary format addresses. Then, accessing a mapped memory from the storage driver in Step 306 includes using the binary format addresses to access ASCII codes stored in the EEPROM.

A first microprocessor using a first operating system is

10   included. Then, the method comprises further steps. Step 301a receives the storage driver as first operating system machine codes. Step 301b stores the storage driver as machine code. Step 301c receives virtual key information as binary format code. Step 301d, using the storage driver, cross-references the virtual key information with EEPROM addresses.

15   Step 301e stores the virtual key information in the EEPROM as machine code.

The three above-described methods all have advantages that must be evaluated in terms the cost of the products, software maintenance, and storage size, and access speed

20

Data Storage Models

For each virtual key event, for example the "record" virtual key (VK_RECORD), there are at least three (3) associated fields to support the HEventRepresentation class. They are: a field for string; a

25   field for symbol; and, a field for color representation. There are several

ways of storing these data. They can be stored in text file format, in text array format, or in a static class format.

Fig. 4a is an example property text file virtual key representation. A test file format implementation takes advantage of

5   java.awt.ResourceBundle class. With a ResourceBundle class, a resource property file can be established to contain VK_RECORD_color, VK_RECORD_string, VK_RECORD_image, VK_RECORD_type variables that link the virtual key representation to HAVi code implementation. Note that the property file contains the HEventRepresentation class's

10   table, which is not part of the API implementation. The property file can· be maintained apart from the API code maintenance.

Fig. 4b depicts the process of using a ResourceBundle to access virtual key (VK) information. A ResourceBundle API specifies a property file and a property attribute (Step 1). The ResourceBundle finds

15   the file and searches for the proper attribute (Step 2). The VK information ("an arrow pointing up") is returned to the ResourceBundle in Step 3, and to the API in Step 4.

Fig. 5 illustrates an example text array virtual key representation. This storage model is a text array embedded table in an

20   API implementation that recommends the format of storage. The data array contains all the fields for the representation of each key, including key code, color, string representation, and image representation file name. Changes in the key representation table must be made by changing the code. On the other hand, the representation can be stored in a smaller

25   memory due to the binary storage format, as compared to the property text file of Fig. 4 using the text format.

Fig. 6 is an example of a static class virtual key representation. This storage model is similar to the text array of Fig. 5 in that the data storage is part of API implementation. Again, any table changes require a change in code. Each field can be accessed

5   programmatically as: EventRepresentation.VK_GO_TO_START.code, EventRepresentation.VK_PAUSE.s, or EventRepresentation.VK_POWER.imgFile, for example.

Data Accessing Models

10   The virtual key information in the above-described data storage models can be accessed in several ways. Each access model uses a different java technology.

Fig. 7 is a representation of a JVM accessing model. The JVM accessing model embeds event representation data in data array (see

15   Fig. 5) or a static class (see Fig. 6). Data is stored using Java data types such as byte arrays, static classes, and static data members. Access data such as virtual key information is just the same as accessing class data members. This access model has a reasonable maintenance cost and reasonable memory cost. The disadvantage of this model is that event

20   representation updates are difficult to perform by a system user.

Fig. 8 is a representation of a Java I/O accessing model. This accessing model stores event representation data in a flash memory in text format. When virtual key information is required, the data is accessed by accessing an attribute in a file via a standard Java class,

25   namely, the ResourceBundle. The ResourceBundle was originally invented by Java platform to support multiple language attributes. One

advantage of this accessing model is that it permits a user to program some virtual key information, as opposed to accepting the de facto settings. For instance, a user can program the key representation of a color key, for example VK_COLOR_0, with a null string setting, to display

5   some user-defined text on this color button. This is done by altering the event representation string field in the flash memory. The modification is permanent until next time the string is altered by user again. This model has high memory requirements, but absolutely no impact to API implementation. It is suitable for high-end feature enriched products.

10   Fig. 9 is a representation of a JNI/storage driver accessing model. This accessing model requires intensive product design and programming, as each event representation field must be allocated an area in a mapped EEPROM memory. The JNI/storage driver model has fixed memory field size and compact data format (such as compressed or

15   binary data). The HAVi L2 GUI access virtual key information via JNI calls to the storage driver. This model has low memory cost, high non-recurring design cost, limited design flexibility, and can implement user programmable features.

Fig. 10 is an example display of virtual keys created with

20   string data and color data. Although not evident from the figure, the CK_0, CK_1, CK_2, and CK_3 are different colors, and the record key (REC) is red.

Fig. 11 is an example display of virtual keys using symbols data. Symbols data can be retrieved, in addition to string and color data,

25   from the HEventRepresentation class by applications. Again, the CK_0,

CK_1, CK_2, and CK_3 are different colors. Other keys may have colors associated with them.

A method for storing and accessing HEventRepresentation virtual key information has been provided. Example of specific storage
5   mediums, protocols, and interfaces have been provided. However, the present invention is not limited to merely these examples. Other variations and embodiments of the invention will occur to those skilled in the art.

10

WE CLAIM:

ATTACHMENT B

```
                    START          S100
                                        101a1

      Receiving VK information as
      Java source code
                                        101a2

      Receiving HEvent Representation application
      as Java source code

      Compiling Java source codes into Jmv    S101b1
      byte codes                              S101b2
                                              101c1

      archiving JMV bytecodes into JAR file

                                              101c2

      accessing a JAR file    S102

                                        104

      retreiving virtual key information


                    Fig. 1
```

START ~200

maintaining HEventRepresentation application in 1st operating system protocol — 201a

Receiving VK information in ResourceBundle property file — 201b

integrating VK information into Table — 201c

Storing table as ResourceBundle property file — 201d

accessing Java I/o ResourceBundle — 202

Retrieving VK information — 204

Fig. 2

START ~ 300

receiving storage driver as machine codes — 301a

Storing storage driver as machine codes — 301b

receiving VK information as binary code — 301c

using storage driver to cross-reference VK information with EEPROM addresses — 301d

storing VK information as machine codes — 301e

calling JNI — 302

using Java byte codes to call storage driver — 304

accessing mapped memory — 306

retrieving VK information — 308
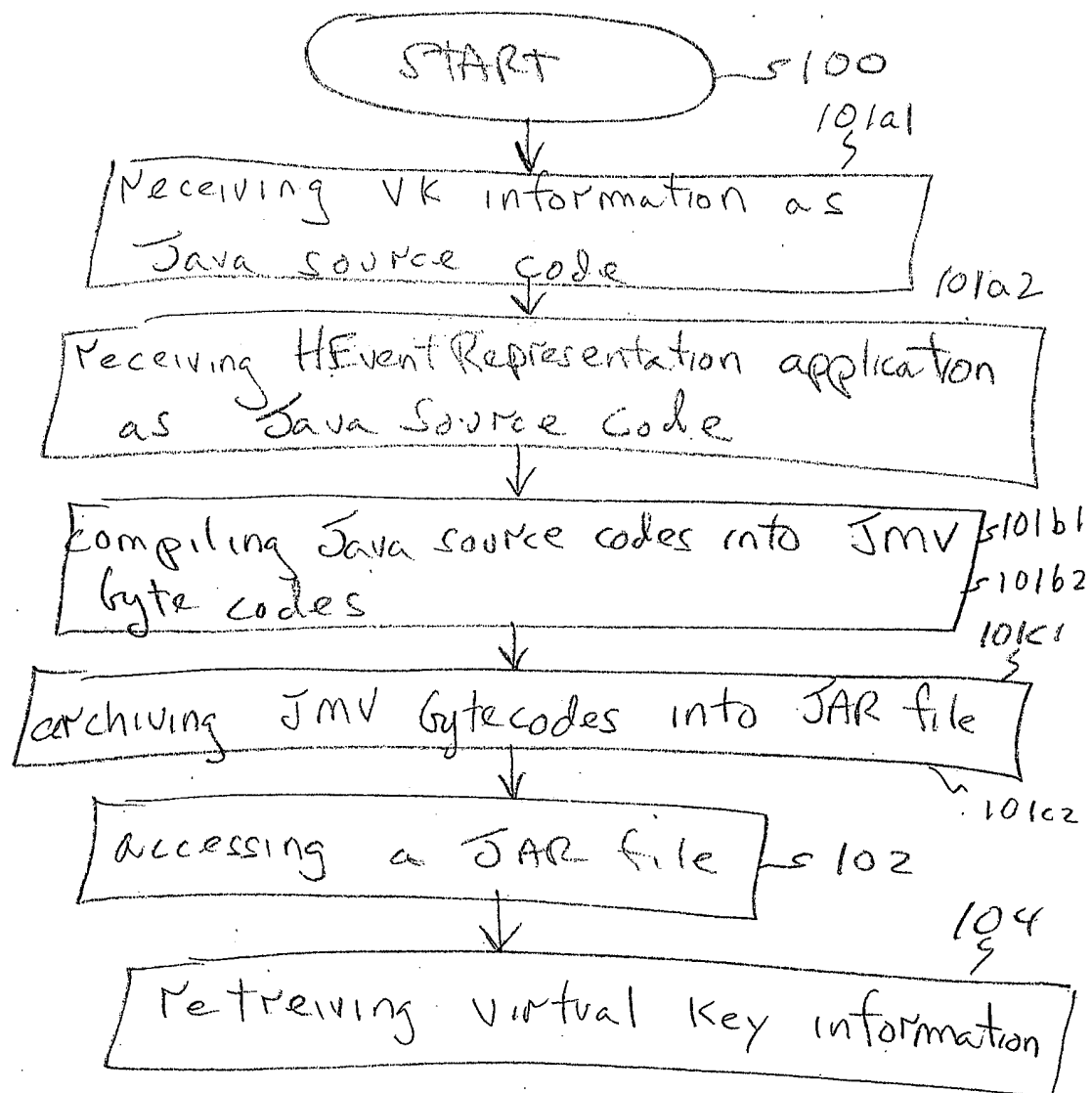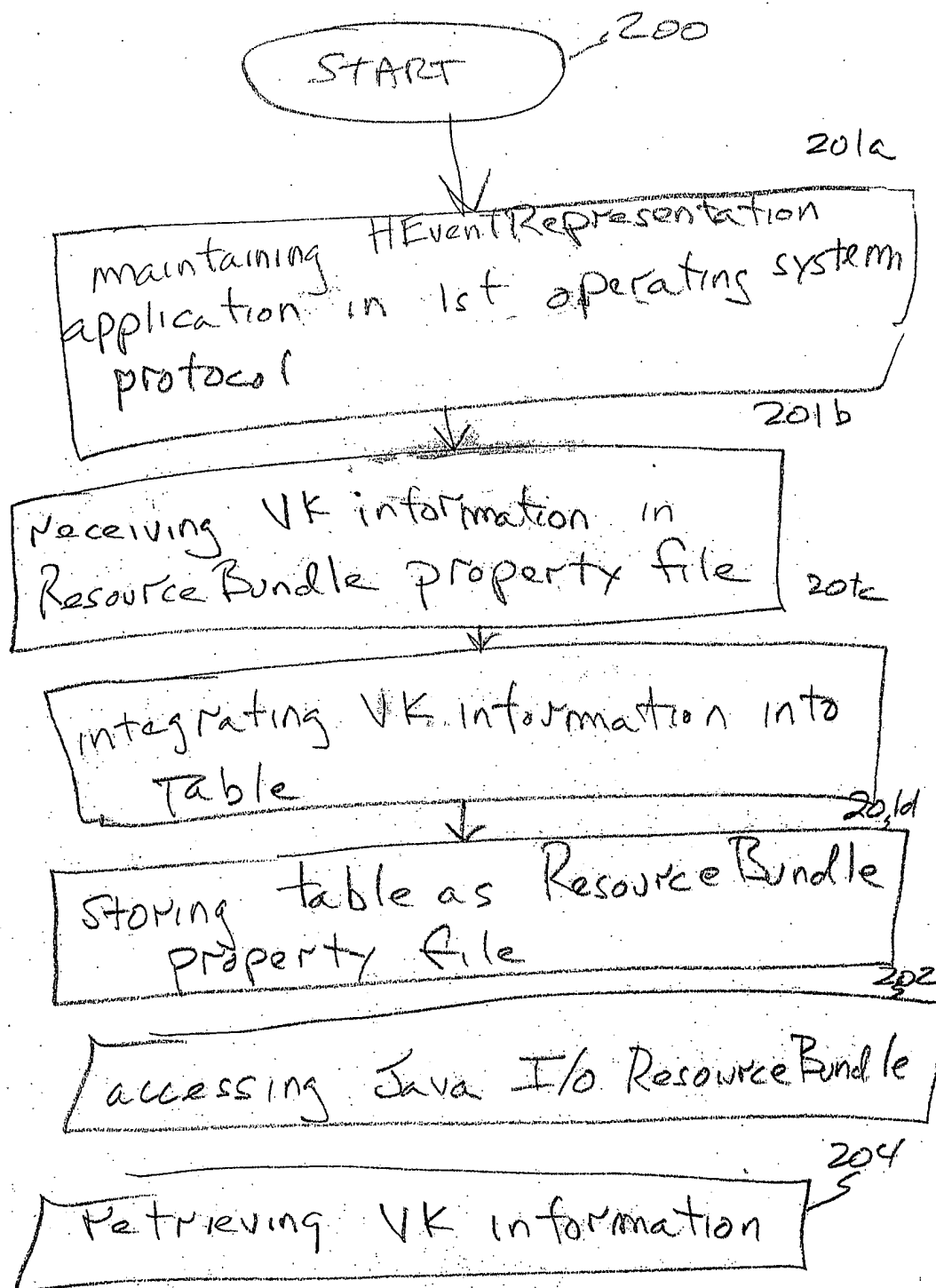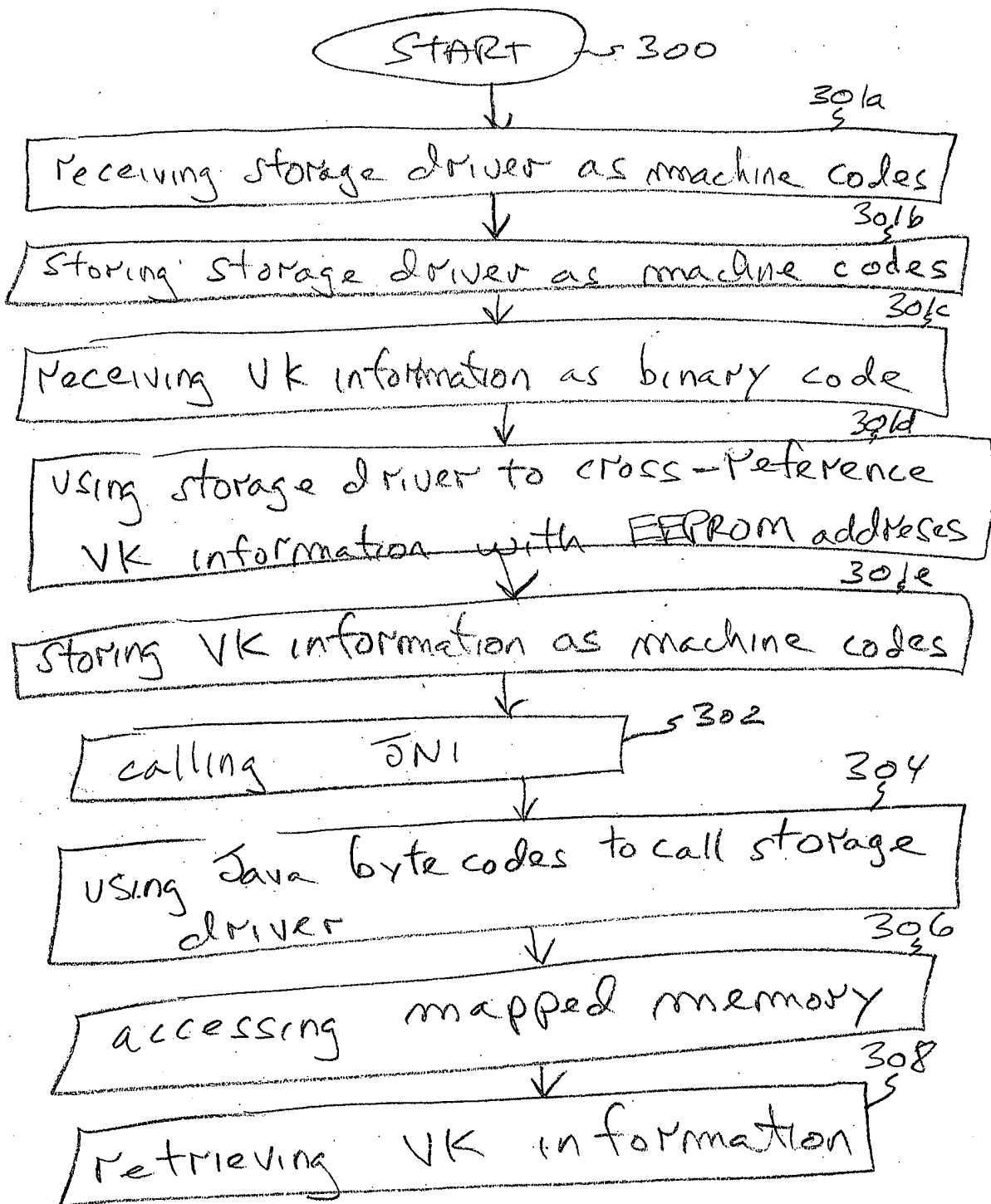
Fig. 3

```
[**platform event representation - color]
VK_GO_TO_START_Color = 21,  21,    21
VK_REWIND_Color      = 0,  0,  0
VK_STOP_Color        = 0,  0,  0
VK_PAUSE_Color       = 0,  0,  0
VK_PLAY_Color        = 0,  0,  0
VK_FAST_FWD_Color    = 0,  0,  0
VK_GO_TO_END_Color   = 0,  0,  0
VK_TRACK_PREV_Color  = 0,  0,  0
VK_TRACK_NEXT_Color  = 0,  0,  0
VK_RECORD_Color      = 245, 0,  0
VK_EJECT_TOGGLE_Color = 0,  0,  0
VK_VOLUME_UP_Color    = 0,  0,  0
VK_VOLUME_DOWN_Color = 0,  0,  0
VK_UP_Color          = 0,  0,  0
VK_DOWN_Color        = 0,  0,  0
VK_LEFT_Color        = 0,  0,  0
VK_RIGHT_Color       = 0,  0,  0
VK_POWER_Color       = 0,  0,  0


[**platform event representation - String]
VK_GO_TO_START_String = Two equilateral triangles, pointing at a line to the left
VK_REWIND_String     = Two equilateral triangles, pointing to the left
VK_STOP_String       = A square
VK_PAUSE_String      = Two vertical lines, side by side
VK_PLAY_String       = One equilateral triangle, pointing to the right
VK_FAST_FWD_String   = Two equilateral triangles, pointing to the right
VK_GO_TO_END_String  = Two equilateral triangles, pointing to a line at the right
VK_TRACK_PREV_String = One equilateral triangle, pointing to a line at the left
VK_TRACK_NEXT_String = One equilateral triangle, pointing to a line at the right
VK_RECORD_String     = A circle, normally red
VK_EJECT_TOGGLE_String = A line under a wide triangle which points up
VK_VOLUME_UP_String   = A ramp, increasing to the right, near a plus sign
VK_VOLUME_DOWN_String = A ramp, increasing to the right, near a minus sign
VK_UP_String         = An arrow pointing up
VK_DOWN_String       = An arrow pointing down
VK_LEFT_String       = An arrow pointing to the left
VK_RIGHT_String      = An arrow pointing to the right
VK_POWER_String      = A circle, broken at the top, with a vertical line in the break


[**platform event representation - image]
VK_GO_TO_START_Image = start.png
VK_REWIND_Image      = rewind.png
VK_STOP_Image        = stop.png
VK_PAUSE_Image       = pause.png
VK_PLAY_Image        = play.png
VK_FAST_FWD_Image    = fastfwd.png
VK_GO_TO_END_Image   = end.png
VK_TRACK_PREV_Image  = prevtrack.png
//VK_TRACK_NEXT_Image = nexttrack1.png
VK_TRACK_NEXT_Image  = nexttrack.png
VK_RECORD_Image      = record.png
VK_EJECT_TOGGLE_Image = eject.png
VK_VOLUME_UP_Image   = volup.png
VK_VOLUME_DOWN_Image = voldown.png
VK_UP_Image          = up.png
VK_DOWN_Image        = down.png
VK_LEFT_Image        = left.png
VK_RIGHT_Image       = right.png
VK_POWER_Image       = power.png


[**platform event representation - type]
VK_GO_TO_START_Type = ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL
VK_REWIND_Type      = ER_TYPE_COLOR | ER_TYPE_SYMBOL
VK_STOP_Type        = ER_TYPE_SYMBOL
VK_PAUSE_Type       = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL
VK_PLAY_Type        = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL
VK_FAST_FWD_Type    = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL
VK_GO_TO_END_Type   = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR |
ER_TYPE_SYMBOL
VK_TRACK_PREV_Type  = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR |
ER_TYPE_SYMBOL
VK_TRACK_NEXT_Type  = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR |
ER_TYPE_SYMBOL
VK_RECORD_Type      =    ER_TYPE_SYMBOL
VK_EJECT_TOGGLE_Type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR |
ER_TYPE_SYMBOL
VK_VOLUME_UP_Type   = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR |
ER_TYPE_SYMBOL
VK_VOLUME_DOWN_Type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR |
ER_TYPE_SYMBOL
VK_UP_Type          = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL
VK_DOWN_Type        = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL
VK_LEFT_Type        = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL
VK_RIGHT_Type       = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL
VK_POWER_Type       = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL
```

Fig. 4a. Property text file to store HEventRepresentation table.

JAVA code using ResourceBundle
API by specifying property file
name (keyProp_EN_US.property,
for instance) and property attribute
to be accessed (VK_UP_String, for
instance)

4

"An arrow pointing up"

1

File: keyProp_EN_US.property
Property: VK_UP_String

ResourceBundle

2

Find file keyProp_EN_US.property
Get property content of
VK_UP_String in file

Property file as shown in "Diagram
1. Property text file to store
HEventRepresentation table"

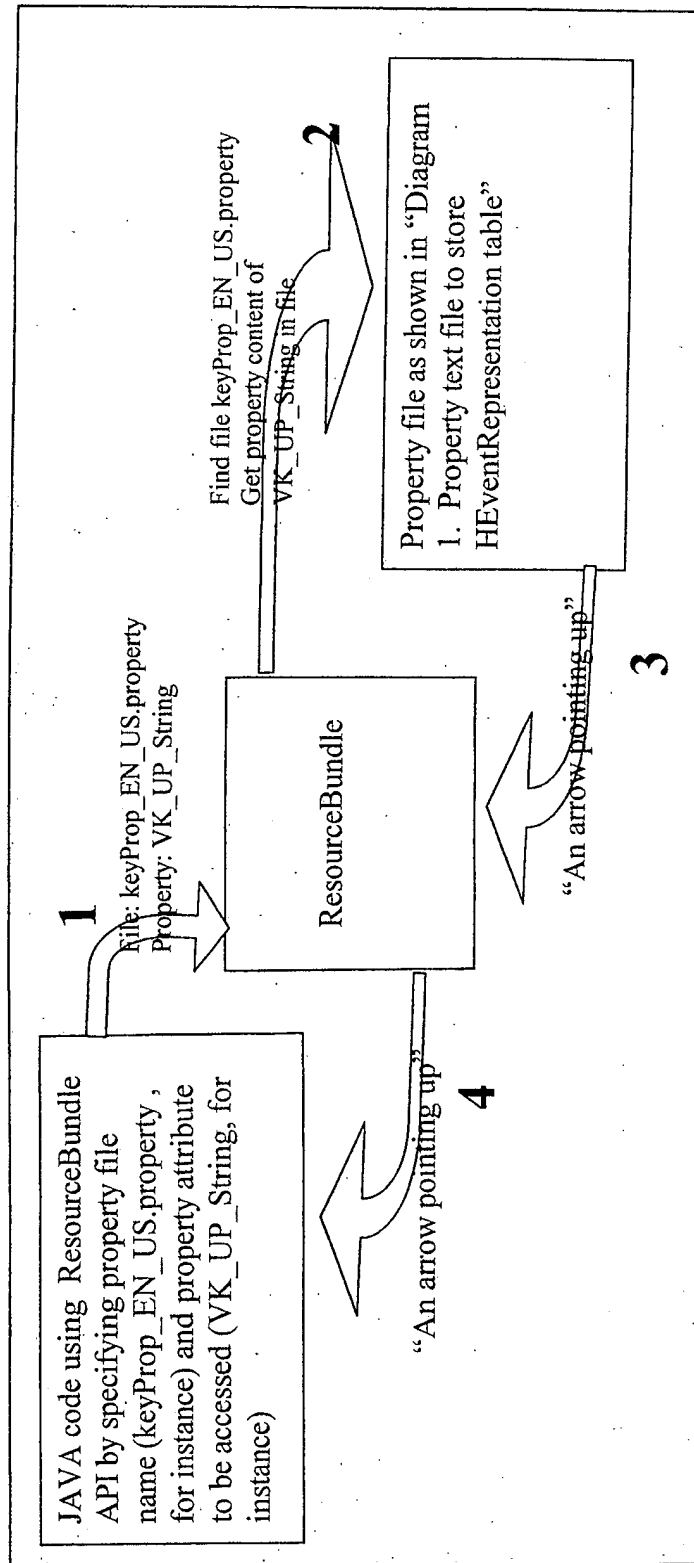**3**

"An arrow pointing up"

Fig. 4b

```
String[]   eventRepresentationData = {
    VK_GO_TO_START, new Color(r, g, b), "|<<", "start.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_ REWIND, new Color(r, g, b), "<<", "rewind.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_ STOP, new Color(r, g, b), "STOP", "stop.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_ PAUSE, new Color(r, g, b), "||", "pause.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_ PLAY, new Color(r, g, b), ">", "play.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_FAST_FWD, new Color(r, g, b), ">>", "fastfwd.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_GO_TO_END new Color(r, g, b), ">>|", "end.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_TRACK_PREV, new Color(r, g, b), "|<", "prevtrack.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_TRACK_NEXT, new Color(r, g, b), ">|", "nexttrack1.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_RECORD, new Color(r, g, b), "O", "record.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_EJECT_TOGGLE, new Color(r, g, b), "EJECT", "eject.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_VOLUME_UP, new Color(r, g, b), "VOL+", "volup.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_VOLUME_DOWN, new Color(r, g, b), "VOL-", "voldown.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_UP, new Color(r, g, b), "^", "up.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_DOWN, new Color(r, g, b), "V", "down.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_LEFT, new Color(r, g, b), "<", "left.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_RIGHT, new Color(r, g, b), "->", "right.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
    VK_GO_TO_POWER, new Color(r, g, b), "0/1", "right.png", ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL,
};
```

Fig. 5   Text array of key event representation

```java
package com.sharplabs.havi.ui.util;

import java.awt.Color;

Public static class EventRepresentationData {
  static class VK_GO_TO_START {
    static final int     code = VK_GO_TO_START;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = "|<<";
    static final String  imgFile = "start.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL;
  }
  static class VK_REWIND {
    static final int     code = VK_REWIND;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = "<<";
    static final String  imgFile = "rewind.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL;
  }
  static class VK_STOP {
    static final int     code = VK_STOP;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = "stop";
    static final String  imgFile = "stop.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL.
  }
  static class VK_PAUSE {
    static final int     code = VK_PAUSE;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = "||";
    static final String  imgFile = "pause.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL;
  }
  static class VK_PLAY {
    static final int     code = VK_PLAY;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = ">";
    static final String  imgFile = "play.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL;
  }
  static class VK_FAST_FWD {
    static final int     code = VK_FAST_FWD;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = ">>";
    static final String  imgFile = "fast_fwd.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL;
  }
  static class VK_GO_TO_END {
    static final int     code = VK_GO_TO_END;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = ">>|";
    static final String  imgFile = "end.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL;
  }
  static class VK_TRACK_PREV {
    static final int     code = VK_TRACK_PREV;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = "|<";
    static final String  imgFile = "prevtrack.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL;
  }
  static class VK_TRACK_NEXT {
    static final int     code = VK_TRACK_NEXT;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = ">|";
    static final String  imgFile = "nexttrack.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL;
  }
  static class VK_RECORD {
    static final int     code = VK_RECORD;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = "REC";
    static final String  imgFile = "record.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL;
  }
  static class VK_EJECT_TOGGLE {
    static final int     code = VK_EJECT_TOGGLE;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = "EJECT";
    static final String  imgFile = "eject.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL;
  }
  static class VK_VOLUME_UP {
    static final int     code = VK_VOLUME_UP;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = "VOL+";
    static final String  imgFile = "volumeup.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL;
  }
  static class VK_VOLUME_DOWN {
    static final int     code = VK_VOLUME_DOWN;
    static final Color      c = new Color(0, 0, 0);
    static final String     s = "VOL-";
    static final String  imgFile = "volumedown.png";
    static final int     type = ER_TYPE_NOT_SUPPORTED | ER_TYPE_STRING | ER_TYPE_COLOR | ER_TYPE_SYMBOL;
  }
  static class VK_UP {
```

Fig. 6 Static class of event representation

Fig. 7　JVM Access Model
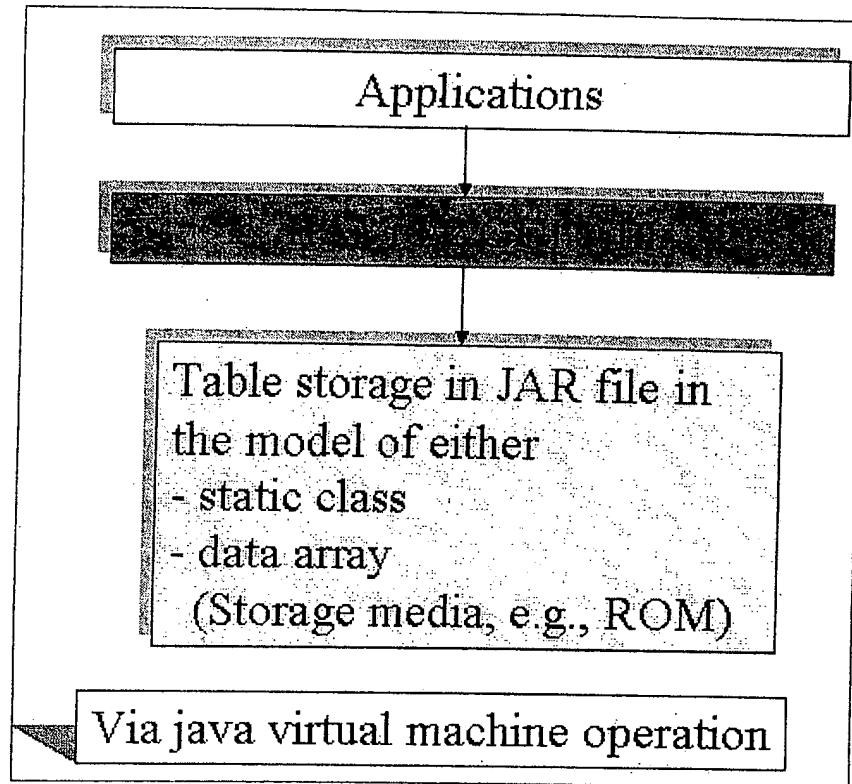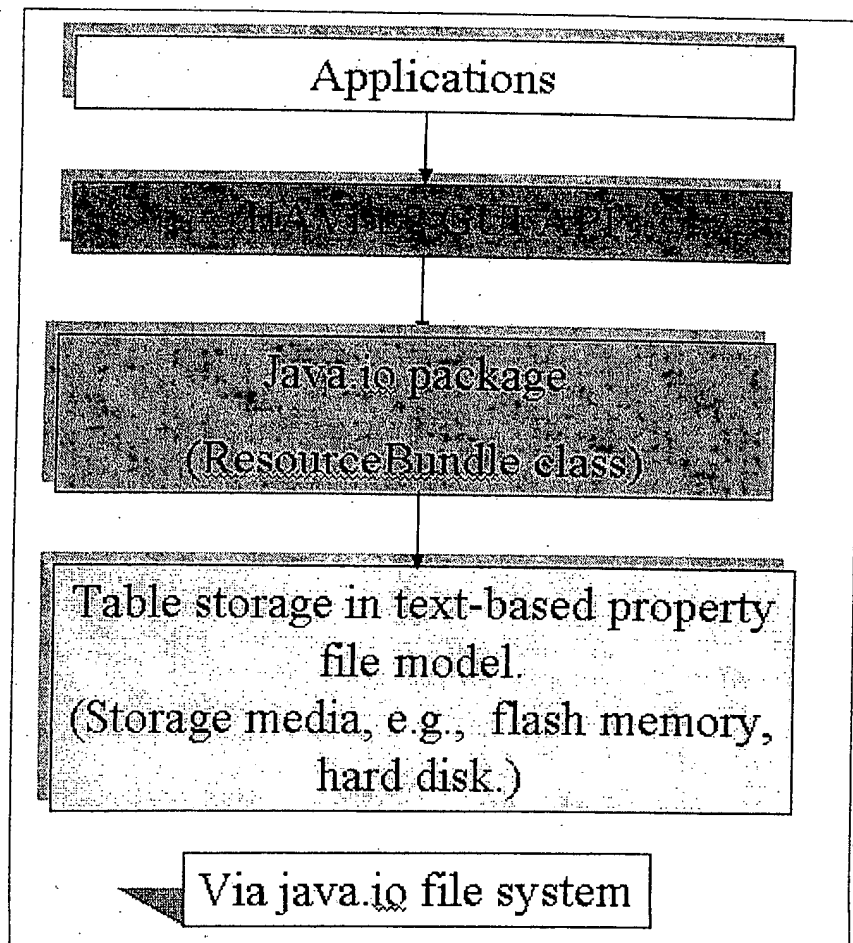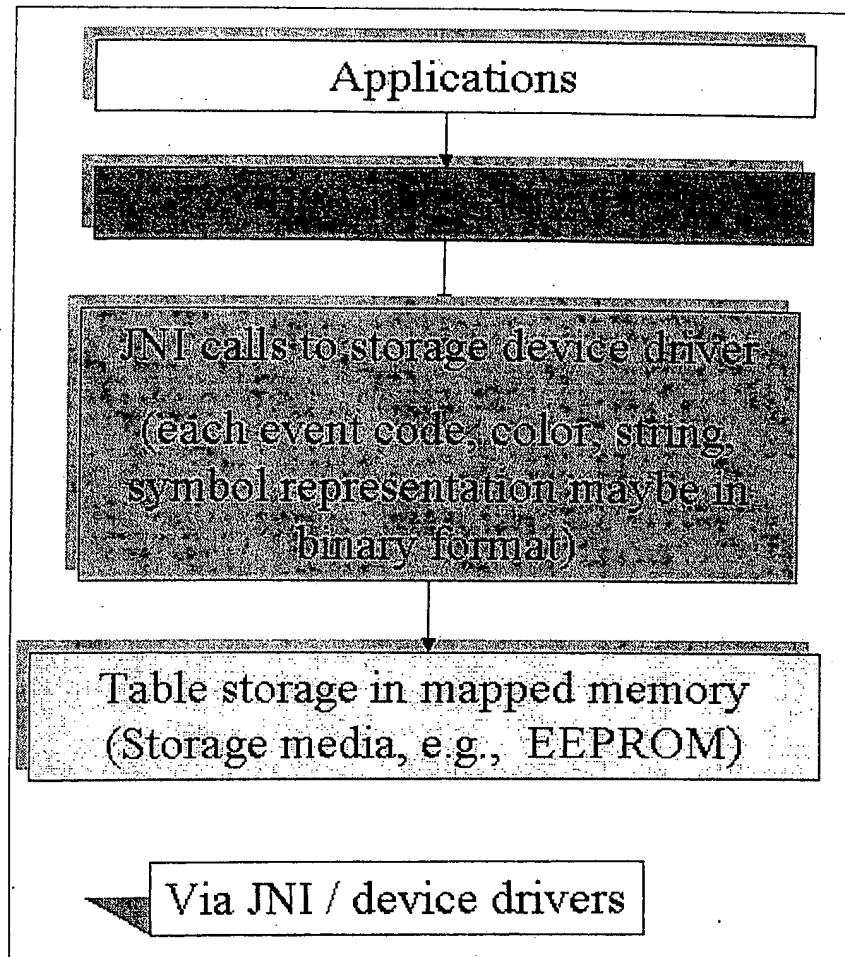
Fig. 8　Java I/O Accessing model

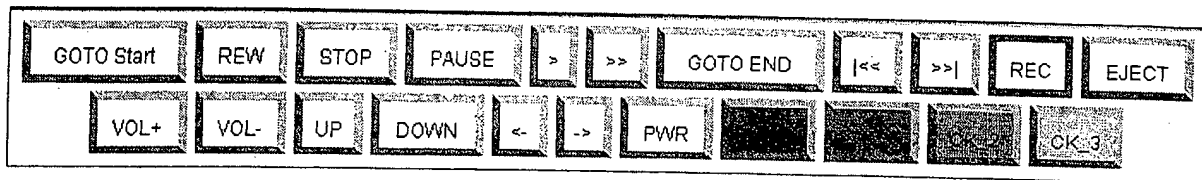Fig. 9   JNI/Storage driver Accessing Model

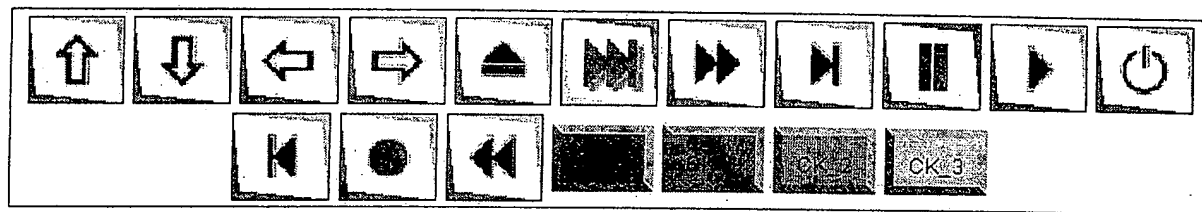Fig. 10   HEventRepresentation using String, Color attribute data

Fig. 11   HEventRepresentation using Symbol, String, and Color
attribute data

ATTACHMENT C

The HAVi Specification

Specification of the Home Audio/Video Interoperability (HAVi) Architecture

HAVi, Inc.

Version 1.1
May 15, 2001

1 General

1.1 Scope

This document provides a specification of the *Home Audio/Video Interoperability Architecture* (also called the *HAVi Architecture*). The HAVi Architecture is intended for implementation on Consumer Electronics (CE) devices and computing devices; it provides a set of services which facilitate interoperability and the development of distributed applications on home networks. HAVi is intended for, but not restricted to, CE devices supporting the IEEE Std 1394-1995 [3] (and future extensions) and IEC 61883 [4] interface standards.

Since a goal of the HAVi Architecture is to be future-proof, interoperability is more than a common command set. HAVi is a software architecture that allows new devices to be integrated into the home network and to offer their services in an open and seamless manner. The HAVi Architecture provides:

- a set of software elements along with the protocols and APIs needed to achieve interoperability

- device abstraction and device control models

- an addressing scheme and lookup service for devices and their resources

- an open execution environment supporting visual presentation and control of devices, and providing runtime support for third party applications

- communication mechanisms for extending the environment dynamically through plug-and-play capabilities

- a versioning mechanism that preserves interoperability as the architecture evolves

- management of isochronous data streams

This document describes the constructs HAVi implements to support interoperability. Specific topics covered include: the system and device models of the HAVi Architecture, the APIs and protocols used by software elements of the HAVi Architecture, and APIs for specific devices. The types of devices supported by HAVi include: tuner, VCR, clock, camera, AV disc, display, amplifier, modem, and Web proxy.

1.2 References

[1] ISO/IEC 13213:1994 Control and Status Register (CSR) Architecture for Microcomputer Buses (IEEE Std 1212-1994).

[2] IEEE P1212 Draft 1.0, Draft Standard for a Control and Status Registers (CSR) Architecture for Microcomputer Buses, October 18, 1999 (approval pending).

[3] IEEE Std 1394-1995, Standard for a High Performance Serial Bus.

[4] IEC 61883 Parts 1 – 5, Standard for a Consumer-Use Digital Interface.

[5] CDR: Object Management Group (OMG) CORBA specification 2.41.

[6] The Java Virtual Machine, Tim Lindholm and Frank Yellin, Addison-Wesley, 1997.

[7] The Java Language Specification, James Gosling, Bill Joy, and Guy Steele, Addison-Wesley, 1996.

[8] Java Development Kit (JDK) 1.1 Core API Specification (http://java.sun.com/products/jdk/1.1/docs/api/packages.html).

[9] PersonalJava 1.1 API Specification (http://java.sun.com/products/personaljava/spec-1-1/pJavaSpec.html).

[10] Portable Network Graphics (PNG) specification v1.0, IETF RFC 2083.

[11] Audio Interchange File Format, version C, allowing for Compression (AIFF-C), Digital Audio Visual Council (DAVIC) 1.3 Part 9, Annex B.

[12] UNICODE Standard Version 2.0.

[13] HAVi Test Requirements, The HAVi, Inc., January 2000.

[14] IEEE std 1394a-2000, Standard for a High Performance Serial Bus – Amendment I

[15] TA Document 1999032: Clarification and Implementation Guideline for Isochronous Connection Management of IEC 61883-1.

[16] HAVi Certification Procedures, The HAVi, Inc. [To be issued]

[17] HAVi Logo Requirements, HAVi, Inc. [To be issued]

[18] JPEG International Standard ISO 10918-1 using JPEG File Interchange Format (JFIF) Version 1.02.

[19] RSA algorithm for digital signature PKCS#1 v2.0, IETF RFC 2437.

[20] SHA-1 Message Digest, specified by National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)." FIPS Publication 180-1, April 17, 1995.

1.3 Terminology

HAVi Acronyms

Acronym	Description
BAV	Base AV device
DCM	Device Control Module
DDI	Data Driven Interaction
FAV	Full AV device
FCM	Functional Component Module
HJA	HAVi Java API
HUID	HAVi Unique Identifier

IAV	Intermediate AV device
LAV	Legacy AV device
SDD	Self Describing Device
SEID	Software Element Identifier

Other Acronyms

Acronym	Description
1394, IEEE 1394	IEEE std 1394-1995
API	Application Programming Interface
AV/C-CTS	Audio/Video Control Command and Transaction Set (specified by the 1394 Trade Association)
AV/C	Audio/Video Control
CDR	Common Data Representation [5]
CE	Consumer Electronics
CTS	Command and Transaction Set
DTV	Digital TV
DV	Digital Video, the consumer version of DVC
DVC	Digital Video Cassette
DVD	Digital Video/Versatile Disc
EPG	Electronic Program Guide
FCP	Function Control Protocol (IEC 61883.1)
GUI	Graphical User Interface
GUID	Global Unique Identifier
IR	infrared
iPCR	input Plug Control Register (IEC 61883.1)
oPCR	output Plug Control Register (IEC 61883.1)
PCR	Plug Control Register (IEC 61883.1)
RTOS	Real-time Operating System
STB	Set-top Box
bslbf	Bit string leftmost bit first
uimsbf	unsigned integer, most significant bit first
UI	User Interface

Definitions

Term	Description
Application Module	An application, of a form defined by HAVi, that may provide a DDI interface and/or means for obtaining havlets.
Application Module code unit	A code unit from which an Application Module is obtained.

attachment	A "stage" of an external connection involving resources within a single device.
base AV device (BAV)	A HAVi-compliant device containing SDD data but not running any of the software elements of the HAVi Architecture.
bytecode	Bytecode for the Java™ virtual machine ("Java bytecode").
client	A software element controlling one or more resources (see also *device resource* and *network resource*).
code unit	Refers to the executable code from which HAVi software elements are obtained. *Native code units* are platform dependent and typically would include machine code for a specific processor. *Java code units* include Java bytecode and so are platform independent. *Embedded code units* are those that are pre-loaded, while *uploadable code units* are those that may be dynamically loaded (and unloaded).
connection	A unidirectional data transfer path created by a HAVi Stream Manager. Typically used for streaming content. Connections originate and/or terminate at functional components. A connection is either an *internal connection* or an *external connection*. *Non-HAVi connections* refer to data transfer paths created by non-HAVi applications or devices.
controller	A device which controls other devices. An IAV or FAV device.
Data Driven Interaction (DDI)	A HAVi mechanism allowing control of software elements (DDI Targets) with access to devices by other software elements (DDI Controllers) with access to user input/output facilities. In this interaction, the user interface is described by a set of data structures (DDI elements) sent between the DDI Controller and the DDI Target.
DDI Controller	A software entity which renders DDI elements and handles user interaction using its (typically local) input/output facilities.
DDI element	The DDI encoding of a user interface element. For example, buttons, icons, sliders, text displays and text entry fields.
DDI protocol	The HAVi messages supporting Data Driven Interaction.
DDI Target	A software entity (typically a DCM) which supports the DDI protocol thereby allowing an associated DDI Controller to use the target's DDI elements to control a device associated with the DDI Target.
device	A physical entity attached to the home network, examples are video players/recorders, cameras, CD and DVD players, set-top boxes, DTV receivers, and PCs.
device connection	An internal connection or an attachment.
device control module (DCM)	A HAVi software element providing an interface for controlling general functions of a device.
device resource	An FCM.
DCM code unit	A code unit from which a DCM is obtained. Installation of a DCM code unit results in one DCM and zero or more FCMs.
embedded code unit	See code unit.
embedded DCM	A DCM pre-loaded on an FAV or IAV. Embedded DCMs typically run on IAV devices and are typically implemented in native code.

4

external connection	A connection where data is transferred across a device boundary.
full AV device (FAV)	A HAVi-compliant device which runs the software elements of the HAVi Architecture including a Java runtime environment.
functional component	An abstraction within the HAVi Architecture that represents a group of related functions associated with a device. For example a DTV receiver may consist of several functional components: tuner, decoder, audio amplifier etc.
functional component module (FCM)	A HAVi software element providing an interface for controlling a specific functional component of a device.
global unique ID (GUID)	A 64-bit quantity used to uniquely identify an IEEE 1394 device. Consists of a 24-bit vendor ID (obtained from the 1394 Registration Authority Committee) and a 40-bit serial number assigned by the device manufacturer. The GUID is stored in a device's configuration ROM and is persistent over 1394 bus resets.
HAVi Architecture	The HAVi Architecture comprises the messaging model, control model, device model, and execution environment defined in this document.
HAVi-compliant device	A device conforming to the HAVi Architecture specification for an FAV, IAV or BAV device.
HAVi Java API	The Java API is defined in this specification.
HAVi Level 1 interoperability	Refers to the features provided by IAVs and embedded DCMs.
HAVi Level 2 interoperability	Refers to the features provided by FAVs and uploaded DCMs.
HAVi RMI	HAVi defined RMI (Remote Method Invocation) based on request and response exchanges among HAVi Messaging Systems.
HAVi unique ID (HUID)	A unique identification of devices and their functional components. Persistent over changes in network configuration (i.e., due to device plug-in or plug-out).
havlet	A HAVi Java application that is uploaded on the request of a controller from a DCM or Application Module.
havlet code unit	A code unit from which a havlet is obtained.
home network	The home network is the generic name used to define the communications infrastructure within the home. This name is used as an abstraction from the physical media and associated protocols. A home network supports both the exchange of control information and the exchange of AV content.
intermediate AV device (IAV)	A HAVi-compliant device which runs the software elements of the HAVi Architecture but does not include a Java runtime environment.
internal connection	A connection where data is transferred within a device.
Java code unit	See code unit.
legacy AV device (LAV)	A non HAVi-compliant device.
native code unit	See code unit.
network resource	IEEE 1394 bandwidth or an IEEE 1394 channel.
scheduled action	A set of operations, involving devices on the home network,

	to be performed at a specific time. For example, tuning to and recording a television program.
SDD data	Self Describing Device (SDD) data is stored in the IEEE 1212 Configuration ROM found on 1394 devices. HAVi specifies Configuration ROM data items that may be used for uploaded DCMs in the HAVi-specified format or for DDI elements in a vendor-specific format.
software element	A HAVi object. A software element responds to a set of messages specified by the API for that element.
software element ID (SEID)	An 80-bit value used to identify software elements. Not guaranteed to be persistent over changes in network configuration (i.e., due to device plug-in or plug-out).
system component	A software element providing basic system services. The system components are: 1394 Communication Media Manager, Messaging System, Event Manager, Registry, DCM Manager, Stream Manager and Resource Manager.
system software element	See system component.
uploaded / uploadable code unit	See code unit.
uploaded / uploadable DCM	A DCM dynamically loaded on an FAV or IAV. HAVi provides support for installing uploadable DCMs implemented in Java bytecode. Installing other forms of uploadable DCMs is vendor dependent.

1.4 Compliance

Each HAVi compliant device (FAV, IAV and BAV) shall:

* support the IEEE1394-1995 and the IEEE1394a-2000 amendment specification.
* provide HAVi SDD data in a IEEE 1212 configuration ROM.
* if the device sources or sinks a stream type for which IEC 61883 transmission has been specified, then the device should support: the PCR and CMP rules for isochronous connections as defined in IEC 61883.1, the CIP protocol as defined in IEC 61883.1, the CIP format specific definition in the corresponding part of IEC 61883.

A HAVi compliant BAV device shall:

* comply to the general HAVi compliance rules described above.
* contain in its HAVi SDD either a signed Java code unit plus corresponding profile, or a reference to a URL containing a signed Java code unit plus corresponding profile.

A HAVi compliant IAV device shall:

* comply to the general HAVi compliance rules described above.
* support IEC 61883.1
* run the following HAVi system components: 1394 Communication Media Manager, Messaging System, Event Manager, and Registry.
* run a DCM Manager if it can host DCMs for LAV or BAV devices.
* run a Stream Manager if software elements on the device require the establishment of streaming connections.
* run a Resource Manager if the device hosts or can host DCMs.
* generate a bus reset if it goes into a standby or low power mode for which the HAVi system is no longer running.
* provide persistent memory for use by HAVi software elements if DCM hosting is supported.

A HAVi compliant FAV device shall:

- comply to the general HAVi compliance rules described above.
- support IEC 61883.1
- run the following HAVi system components: 1394 Communication Media Manager, Messaging System, Event Manager, Registry, DCM Manager, Stream Manager and Resource Manager.
- generate a bus reset if it goes into a standby or low power mode for which the HAVi system is no longer running.
- provide persistent memory for use by HAVi software elements.
- run a Java runtime environment.
- implement the HAVi Java APIs as listed in Chapter 7.
- run a DCM representing itself.

A HAVi compliant bytecode DCM shall:

- refer only to classes specified in HAVi FAV profile #1: *DCM and Application Modules*, or implemented by the DCM itself.

A HAVi compliant bytecode Application Module shall:

- refer only to classes specified in HAVi FAV profile #1: *DCM and Application Modules*, or implemented by the Application Module itself.

A HAVi compliant havlet shall:

- refer only to classes specified in HAVi FAV profile #2: *Havlets*, or implemented by the havlet itself.

A set of test requirements for HAVi devices can be found in [13].

2 Overview

The HAVi Architecture specifies a set of Application Programming Interfaces (APIs) allowing consumer electronics manufacturers and third parties to develop applications for the home network. Thus the home network is viewed as a distributed computing platform, and the primary goal of the HAVi Architecture is to assure that products from different vendors can *interoperate* – i.e., can cooperate to perform application tasks.

To explain fully the interoperability aspects of the architecture, it is necessary to begin with an overview of home networking and identify the requirements addressed by the HAVi Architecture.

2.1 The Home Network

Current CE devices, such as DVD players and DV camcorders, are sophisticated digital processing and digital storage systems. By connecting these devices in networks, it is possible to share processing and storage resources – this allows new applications that:

▦ coordinate the control of several CE devices simultaneously, and

▦ simplify operation of devices by the user.

For instance one device may initiate recording on a second while accessing EPG (Electronic Program Guide) information on a third. The home network provides the fabric for connecting CE devices. It allows connected devices to exchange both control information (one device sending a command to another) and AV content (one device sending an audio or video stream to another). To be successful in the consumer electronics domain the home network must meet several requirements. These include: timely transfer of high-data-rate AV streams, self-configuration and self-management, hot plug-and-play, and low-cost cabling and interfaces. The HAVi Architecture is intended for networks based on the IEEE 1394 standard. 1394 is a powerful technology that meets many of the requirements of home networks. An example of a 1394 network is shown below:
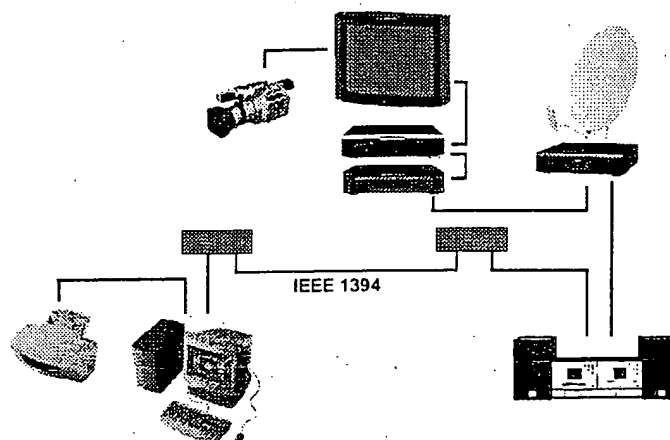
Figure 1.   A 1394 Network with AV Clusters

The underlying structure for the home network consists of interconnected clusters of devices. Typically, there will be several clusters in the home, with one per floor or one per room. Each cluster will work as a set of interconnected devices to provide services to users. Often one device will control other devices. However, the HAVi Architecture is sufficiently flexible to allow home networks with no single master control device.

2.2 Requirements

The HAVi Architecture is an open, light-weight, platform-independent and architecturally neutral specification that allows consumer electronics manufacturers to develop interoperable devices, and independent application developers to write applications for these devices. It can be implemented on different hardware/software platforms and does not include features that are unique to any one platform.

The interoperability interfaces of the HAVi Architecture are extensible and can be advanced as market requirements and technology change. They provide the infrastructure to control the routing and processing of isochronous and time-sensitive data such as audio and video content.

2.2.1 Legacy Device Support

The HAVi Architecture supports legacy devices, i.e., devices that already exist and are available to users. This is important since the transition to networked devices is going to be gradual – with manufacturers not suddenly producing only networked devices and consumers not suddenly replacing their existing devices.

Legacy devices can also be characterized by the degree to which they support 1394 and industry standard protocols for 1394 such as IEC 61883. In particular, legacy devices can be divided into the following categories:

※ non-1394 devices

※ 1394 devices not supporting the HAVi Architecture

Most existing CE devices fall into the first category, while existing devices with 1394 interfaces fall into the second category.

HAVi-compliant devices, as opposed to legacy devices, are those that support the HAVi Architecture. The various categories of HAVi-compliant devices are described in section 2.3.3.

2.2.2 Future-Proof Support

The CE industry has great concern that new products work with existing products. While currently this is largely a question of media formats and interconnect standards, the HAVi Architecture supports future devices and protocols through several software-based mechanisms. These include:

※ persistent device-resident information describing capabilities of devices

※ a write-once, run-everywhere language (*Java*), used for software extensions

※ a device independent representation of user interface elements

Each HAVi-compliant device may contain persistent data concerning its user interface and device control capabilities. This information can include Java bytecode that can be uploaded and executed by other devices on the home network. As manufacturers introduce new models with new features they can modify the bytecode shipped with the device. The new functionality added to the bytecode mirrors the new features provided by the device. Similarly new user interface elements can be added to the stored UI representation on the device.

2.2.3 Plug-and-Play Support

Home network consumer devices are easy to install, and provide a significant portion of their value to the consumer without any action on the user's part, beyond physically connecting the cables. This is in distinction to existing devices that require configuration to provide some major portion of their functionality. Home networking technology offers "hot" plug-and-play (not requiring the user to switch off devices), and safe and reliable connections.

In the HAVi Architecture, a device configures itself, and integrates itself into the home network, without user intervention. Low-level communication services provide notification when a new device is identified on the network.

While there will often be settings the user may change to suit his or her preferences, the HAVi Architecture does not require the user to perform any additional installation operations (as compared to installation for non-networked or stand-alone usage). Frequently, installing a device on the home network will be simpler than stand-alone installation since new devices can obtain configuration information from those already on the network. Thus the infamous "flashing clock on the VCR" can be solved by having the VCR set its clock to that of another device on the network – for example a DTV receiving time signals via digital broadcast.

2.2.4 Flexibility

The HAVi Architecture allows devices to present multiple user interfaces, adapting to both the

10

user's needs and the manufacturer's need for brand differentiation. The architecture includes a flexible device model that scales gracefully from simple CE devices like a CD player or audio amplifier to resource-rich, intelligent devices such as DTV receivers.

2.3 System Model

2.3.1 Control Model

The home network is considered to consist of a set of AV devices. Each device has, as a minimum, enough functionality to allow it to communicate with other devices in the system (with the exception of legacy devices, see section 2.3.3).

During the course of interaction, devices may exchange control information and data in a peer-to-peer fashion. This ensures that, at the communication level, no one device is required to act as a master or controller for the system. However, it also allows a logical master or controller to impose a control structure on the basic peer-to-peer communication model.

The HAVi control model makes a distinction between *controllers* and *controlled devices*. A controller is a device that acts as a host for a controlled device. A controlled device and its controller may reside on the same physical device or on separate physical devices.

In terms of the HAVi control model, a controller is said to host a *Device Control Module* (DCM) for the controlled device. The control interface is exposed via the API of this DCM. This API is the only access point for applications to control the device.

For instance, an intelligent television in the family room might be the controller for a number of interconnected devices. A controlled device could contain Java bytecode that constructs a user interface for the device and allows external control of the device. When these devices are first connected, the controller obtains the user interface and control code. An icon representing the device may then appear on the television screen, and manipulating the icon may cause elements of the control program to actuate the represented device or devices in prescribed ways.

The home network allows a single device, or a group of devices communicating amongst themselves, to deliver a service to a user or an application. When it is necessary for a device to interact with a user, a GUI for the device may be presented on a device with display capabilities (possibly the device in question or possibly a different device).

.DCMs are a central concept to the HAVi architecture and the source of flexibility in accommodating new devices and features. DCMs can be distinguished in several ways (see the figure below).
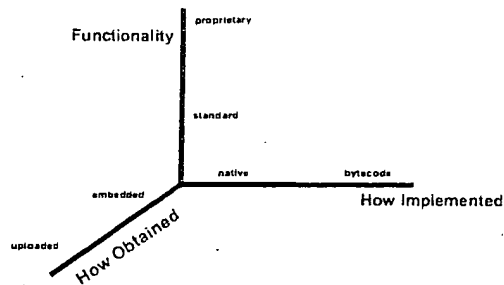
Figure 2.   DCM Characteristics

The first DCM characteristic is how the DCM is obtained by the controller:

▨   *embedded DCM* – a DCM that is part of the resident software on a controller.

▨   *uploaded DCM* – a DCM that is obtained from some source external to the controller and dynamically added to the software on the controller.

The second characteristic is whether a DCM is platform (controller) dependent or platform independent:

▨   *native DCM* – a DCM that is implemented for a specific platform, it may include machine code for a specific processor or access platform specific APIs.

▨   *bytecode DCM* – a DCM that is implemented in Java bytecode.

Finally, DCMs can be distinguished by their functionality (or, conversely, their range of use):

▨   *standard DCM* – a DCM that provides the standard HAVi APIs. Such a DCM provides basic functionality but is able to control a wide range of devices.

▨   *proprietary DCM* – a DCM that provides vendor-specific APIs (in addition to the standard HAVi APIs). Such a DCM would offer additional features and capabilities over a standard DCM but could control a narrower range of devices, perhaps only a specific device or model.

HAVi provides support for uploaded DCMs written in Java bytecode so, in the remainder of this document, "uploadable DCM" implies a bytecode DCM unless indicated otherwise. (Note – native uploaded DCMs are possible but are beyond the scope of this document since their installation requires vendor-specific extensions to HAVi.)

2.3.2 Device Model

A distinction is made between *devices* and *functional components*. A good example of this distinction can be found in a normal TV set. Although the TV set is generally one physical box, it contains several distinct controllable entities, e.g. the tuner, display, audio amplifier, etc. The controllable entities within a device are called functional components.

2.3.3 Device Classification

HAVi classifies CE devices into four categories: Full AV devices (FAV), Intermediate AV devices (IAV), Base AV devices (BAV), and Legacy AV devices (LAV). HAVi-compliant devices are those in the first three categories, all other CE devices fall into the fourth category. Referring to the distinction between controllers and controlled devices – FAVs and IAVs are controllers while BAVs and LAVs are controlled devices. The presentation associated with products of the various categories in the marketplace is defined in the HAVi Logo Requirements document [17].

2.3.3.1 Full AV Devices

A Full AV device contains a complete set of the software elements comprising the HAVi Architecture (see section 2.4.4). This device class generally has a rich set of resources and is capable of supporting a complex software environment. The primary distinguishing feature of an FAV is the presence of a runtime environment for Java bytecode. This allows an FAV to upload bytecode from other devices and so provide enhanced capabilities for their control. Likely candidates for FAV devices would be Set Top Boxes (STB), Digital TV receivers (DTV), general purpose home control devices, and even Home PC's.

2.3.3.2 Intermediate AV Devices

Intermediate AV devices are generally lower in cost than FAV devices and more limited in resources. They do not provide a runtime environment for Java bytecode and so cannot act as controllers for arbitrary devices within the home network. However an IAV may provide native support for control of particular devices on the home network.

2.3.3.3 Base AV Devices

These are devices that, for business or resource reasons, choose to implement future-proof behavior by providing uploadable Java bytecode, but do not host any of the software elements of the HAVi Architecture. These devices can be controlled by an FAV device via the uploadable bytecode or from an IAV device via native code. The protocol between the BAV and its controller may or may not be proprietary. Communication between a FAV or IAV device and a BAV device requires that HAVi commands be translated to and from the command protocol used by the BAV device.

2.3.3.4 Legacy AV Devices

LAV devices are devices that are not aware of the HAVi Architecture. These devices use proprietary protocols for their control, and quite frequently have simple control-only protocols. Such devices can work in the home network but require that FAV or IAV devices act as a gateway. Communication between a FAV or IAV device and legacy device requires that HAVi commands be translated to and from the legacy command protocol.

2.4 HAVi Software Architecture

2.4.1 Object-Based

Services in the HAVi Architecture are modeled as objects. Each object is a self-contained entity, called a *software element*, accessible through a well-defined interface and executing within a software execution environment hosted by the device on which the object runs. Note that different

devices may host different execution environments. Services are accessed, using the communications infrastructure, via their well-defined interfaces.

Services in the HAVi Architecture can be provided by device manufacturers, or can be added by third party vendors. The software model makes no distinction between "standard" services and vendor services; they are both implemented as objects.

2.4.2 Software Element Identifiers

Each object is uniquely named. No distinction is made between objects used to build system services and those used for application services. Objects make themselves known via a system wide naming service known as the *Registry*.

Objects in the system can query the Registry to find other objects and can use the result of that query to send messages to those objects.

The identifier assigned to an object is created by the *Messaging System* before an object registers. These identifiers are referred to as SEIDs – *Software Element Identifiers*. SEIDs are guaranteed to be unique, however the SEID assigned to an object may change as a result of reconfiguration of the home network (for example, device plug-in or removal, or re-initialization of a HAVi device).

2.4.3 Message-Based Communication

All objects communicate using a message passing model. Any object that wishes to use the service of another object does so by using a general purpose message passing mechanism that delivers the service request to the target object. The target object is specified using the unique SEID discussed above.

This general purpose message passing mechanism abstracts from the details of physical location, i.e. there is no distinction between an object on the same device and one on a remote device. The actual implementation of the message passing mechanism will differ from device to device and between vendors. However, the format of HAVi messages, and the protocol used for their delivery, must be common so that interoperability is assured.

The general intent of the object model and Messaging System is to provide a completely generic software model that is sufficiently flexible to allow multiple implementations with a variety of software systems and languages. Details of the binding between messages and the code that handles them are left to the system implementor.

2.4.4 Software Elements

The software elements of the HAVi Architecture support the basic notions of network management, device abstraction, inter-device communication, and device user interface (UI) management. Collectively these software elements expose the *Interoperability API*, a set of services for building portable distributed applications on the home network. The software elements themselves reside above a vendor specific platform such as a real-time operating system (RTOS). The diagram below depicts the arrangement of software elements on an FAV device. While not intended as an implementation blueprint, the diagram does highlight how the HAVi software elements form a middle layer between platform specific APIs and platform independent applications.
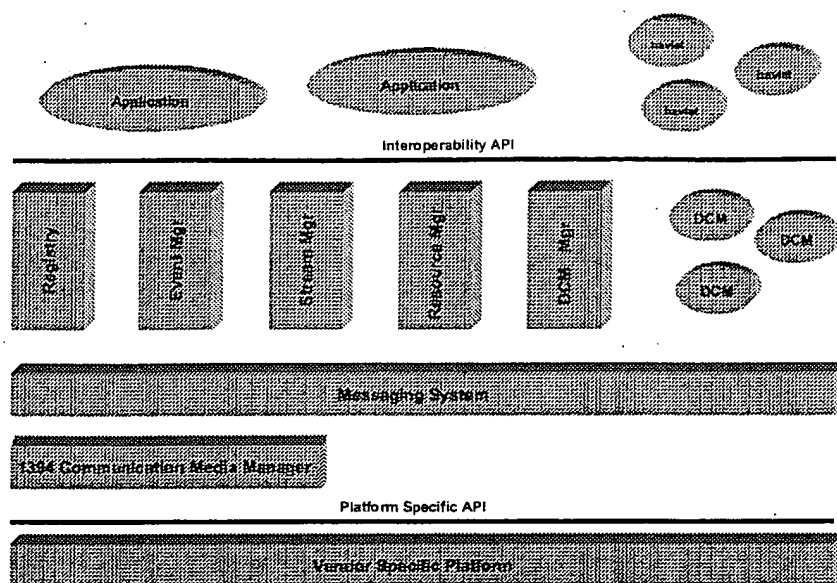
Figure 3. HAVi Architectural Diagram (FAV)

The software elements comprising the HAVi Architecture and defined in this specification are:

- *1394 Communication Media Manager* – allows other software elements to perform asynchronous and isochronous communication over 1394.

- *Messaging System* – responsible for passing messages between software elements.

- *Registry* – serves as a directory service, allows any object to locate another object on the home network.

- *Event Manager* – serves as an event delivery service. An event is the change in state of an object or of the home network.

- *Stream Manager* – responsible for managing real-time transfer of AV and other media between functional components.

- *Resource Manager* – facilitates sharing of resources and scheduling of actions.

- *Device Control Module* (DCM) – a software element used to control a device. DCMs are obtained from *DCM code units*. Within a DCM code unit are code for the DCM itself plus code for *Functional Component Modules* (FCMs) for each functional component within the device. In addition a DCM code unit may include a *havlet* allowing user control of the device and its functional components.

- *DCM Manager* – responsible for installing and removing DCM code units on FAV and IAV devices.

In addition to the above software elements specified by the HAVi Architecture, devices on the home network may contain the following:

▓ *Application Module* – The HAVi architecture provides a general platform for several forms of applications. In general a *HAVi application* is one that creates software elements that use other software elements to provide specific services. A HAVi application may be developed in native code and embedded (resident) on an FAV or IAV. It is also possible for a HAVi application to be in the form of Java bytecode and obtained from external sources (i.e., uploaded over the Internet) or from existing software elements using mechanisms defined by HAVi. In particular, an Application Module is a software element that may provide a DDI interface and/or a havlet.

▓ *Self Describing Device (SDD) data* – HAVi-compliant devices contain descriptive information about the device and its capabilities. This information, called SDD data, follows the IEEE 1212 addressing scheme used for Configuration ROM. The SDD data may include a DCM code unit and vendor-specific data for constructing user interface elements.

▓ *Java Runtime Environment* – provides an execution environment for uploaded DCMs and applications implemented using Java bytecode.

▓ *DDI Controller* – a software element involved with user interaction. The DDI (Data Driven Interaction) Controller handles user input and interprets (renders) DDI elements.

The following table summarizes which architectural elements are present for the various device categories, which are absent and which are optional. An optional element is indicated by "[]", a "check mark" indicates that the element is present on the device itself with the following provisions:

▓ A DDI Controller is required on an IAV or FAV if the device will render DDI elements according to the DDI protocol. Display-capable IAVs or FAVs must contain DDI Controllers.

▓ A Resource Manager is required on an IAV device that hosts or can host DCMs.

▓ A Stream Manager is required on an IAV if applications on the IAV will make streaming connections.

▓ A DCM Manager is required on an IAV device if it can host DCMs for BAVs or LAVs on the 1394 network.

▓ For both BAV and LAV devices there must be an associated device control module somewhere on the home network. For a BAV device, the DCM is typically obtained via the device's SDD data. For LAV devices, the DCM may be embedded on the controlling FAV or IAV.

▓ For IAV devices it is not necessary that a DCM exists on the home network. (However, if such a DCM does not exist then interoperable applications cannot control the device.)

▓ A DCM for an FAV or IAV resides on the device itself – i.e., an IAV (if it has a DCM) or an FAV hosts its own DCM.

Table 1.   HAVi Configurations

Device Class / Element	FAV	IAV	BAV	LAV
Java Runtime	✔			
Application Module	[✔]	[✔]		
DDI Controller	[✔]	[✔]		
Resource Manager	✔	[✔]		
Stream Manager	✔	[✔]		
DCM Manager	✔	[✔]		
Registry	✔	✔		
Event Manager	✔	✔		
Messaging System	✔	✔		
1394 Communication Media Manager	✔	✔		
SDD data	✔	✔	✔	
DCM	✔	[✔]	✔	✔

2.5 User Interface Support

The primary goal of the user interface of the home network is to offer users an easy to use operating environment. The HAVi Architecture allows users to control devices through familiar means, such as via the front panel or via the buttons of a remote controller. In addition the HAVi Architecture allows device manufacturers to specify graphical user interfaces (GUIs) which can be rendered on a range of displays varying from text-only to high-level graphical displays. The GUI need not appear on the device itself, it may be displayed on another device and the display device may potentially be from another manufacturer. To support this powerful feature, the HAVi Architecture provides two mechanisms – the first, the *Level 1 UI*, is intended for IAVs and is called *Data Driven Interaction* (DDI), the second, the *Level 2 UI*, consists of Java APIs that may be provided by FAVs.

2.5.1 Level 1 UI

In essence, SDD data may include, in a vendor dependent manner, *DDI elements* – a platform independent encoding of user interface elements. DDI elements can be loaded from a *DDI Target*, typically a DCM, and displayed by a *DDI Controller*. The DDI Controller retrieves the DDI elements via the DCM (rather than directly from the SDD data, so it is possible that the DCM itself is the source of DDI elements). The DDI Controller generates HAVi messages in response to user input, it also responds to HAVi messages sent by the DCM as a result of changes in device state. This communication is called the *DDI protocol*.

It should be emphasized that the DDI Controller does not understand what happens as a result of issuing or responding to a control message. The DDI protocol involves only abstractions of user interface elements and user actions and is independent of any particular device semantics. Therefore, it is possible for a DDI Controller to handle new device functions which were not known at the time of DDI Controller implementation.

The DDI Controller cannot provide guarantees over the graphical rendition of DDI elements actually presented to the user, since their representation may be changed due to lack of display screen space or other Controller resource limitations. (Furthermore, application software can create different representations, using the DDI elements as "hints".) However the DDI Controller is required to try to preserve the appearance of DDI elements subject to its rendering capabilities.

2.5.1.1   Layout Mechanism

Layout rules of DDI elements are based on geometric coordinates and use x, y values for each DDI element. DDI elements are arranged in a hierarchy and positioned relative to their parents. The top level of hierarchy is a DDI *panel*.

The DDI Target *suggests* a preferred layout, which is encoded into the DDI data structure. However, the DDI Controller may tailor the presentation of DDI elements based on its own limitations, such as screen size, ability to display graphics or text only, etc.

2.5.1.2   Navigation Mechanism

The navigation between DDI elements within the same panel is handled locally by the DDI Controller. The DDI Target may suggest navigation rules between certain DDI elements. Because such navigation rules are just suggestions, the controller may tailor the navigation of the display based on its adjustment of DDI layout.

2.5.2 Level 2 UI

A Level 2 user interface is constructed by bytecode applications running on FAVs. The Java APIs used for implementing a Level 2 UI are based on a subset of Java AWT 1.1 and the following extensions specified by HAVi:

* support for different pixel aspect ratios, screen aspect ratios and screen sizes
* support for alpha blending and video / image layering
* support for remote control input
* support for a set of visual interface components patterned after the features offered by the Level 1 DDI elements

2.5.3 User Notification

There are various events generated by HAVi software elements that must come to the attention of the users of the HAVi system in order that they may react to the situations signaled by these events. Examples include the events generated by the Resource Manager, Stream Manager or Registry. Furthermore, applications and devices on the home network may be presented visually to the user, and information presented to the user should accurately reflect the state of applications and devices. Consistent presentation of state is particularly important since: 1) a device may be acted on in several ways: via its front panel and via a Level 1, Level 2, or arbitrary application accessing its DCM; and 2) several users may simultaneously access or view the same device. For BAV and LAV devices an (icon) representation can be presented to the user. Through such a representation, a user may request to use or manipulate a device or its data.

In order to disseminate information on system state to all users, it is recommended that the notifications of HAVi events be presented (in a vendor dependent way) on all display-capable FAV and LAV controllers in the network. There may be (vendor dependent) facilities allowing the user to disable such notifications. Annex 11.9 lists the events defined by HAVi. Certain events, such as those generated when installing or uninstalling DCMs can be used to detect the addition or removal

of devices and so can help provide a global view of the home network. Other events can be used to relay to the user information concerning the operating state of devices. Still other events can be used to inform the user of anomalous conditions such as communication failure. And yet other events can be used to inform the user of the failure to complete a previously scheduled activity (such as a planned recording at a future date).

2.6 Home Network Configurations

The HAVi Architecture defines how devices are abstracted within the home network and establishes a framework for device control. It defines APIs and messaging protocols so that interoperability is assured, and it defines how future devices and services can be integrated into the architecture. The HAVi Architecture makes no restrictions, however, on what types of devices must be present in the home network. As a result several configurations are possible – networks without FAV devices, networks with multiple FAV devices, networks with LAV and BAV devices only, etc. Depending upon the types of devices on the home network, several different operational configurations are possible.

2.6.1 LAV and BAV Only

The HAVi Architecture does not provide any support for networks consisting of only BAV and LAV devices. However with the addition of a HAVi controller (an IAV or FAV) to the network, these devices can be made available to applications.

2.6.2 IAV or FAV as Controller

IAV and FAV devices act as controllers for the other device classes and provide a platform for the system services comprising the HAVi Architecture. To achieve this, FAVs may host Java bytecode DCMs while IAVs may host embedded DCMs. From an interoperability perspective, the primary role of a controller is to provide a runtime environment for DCMs. Applications use the APIs provided by the DCM to access the controlled device.

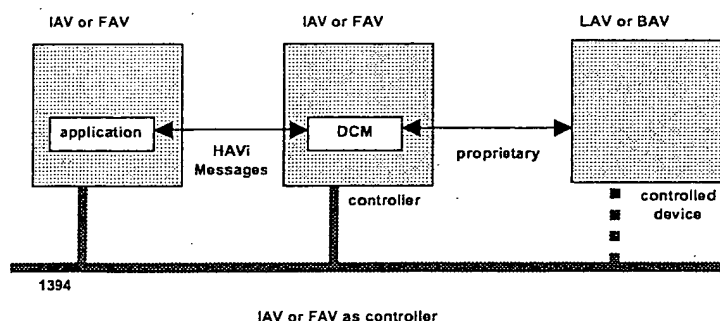Figure 4.   HAVi Controllers

2.6.3 IAV or FAV as Display

Generally, IAVs and FAVs will have an associated display device that is used for display of AV content and GUIs. However, the architecture does not mandate this and an IAV or FAV device

may be "headless" (without display capability). In this case they will cooperate with other IAV or FAV devices with display capability. A display capable IAV is required to support a DDI Controller. A display capable FAV is required to support a DDI Controller and a Level 2 UI. Proprietary low-level graphic manipulation APIs can be used by the DDI Controller to access the display itself, but these interfaces are not exposed as part of the Interoperability APIs.

IAV or FAV as display device
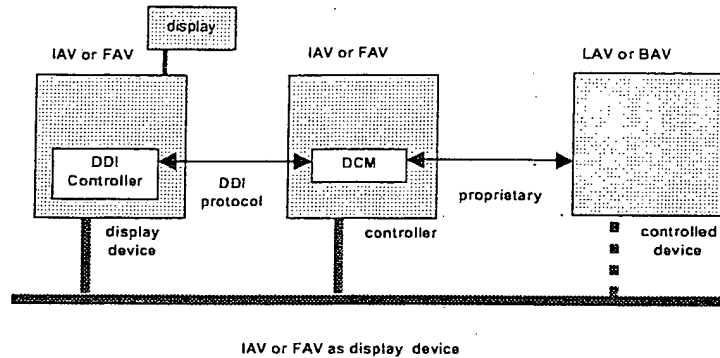
Figure 5. HAVi Displays

2.6.4 Peer-to-Peer Architecture between FAVs and IAVs

In a home network, there may be more than one FAV or more than one IAV. In this case, each controller (IAV or FAV) cooperates with other controllers to ensure that services are provided to the user. This allows devices to share resources. An example is described in section 2.6.3 where a device without display capabilities uses a remote device to display DCM user interfaces. A more elaborate example could be an FAV device utilizing the services of a data conversion module located on a remote device to set up a data stream between two AV devices.

2.6.5 IAV as Controller and Display

A home network may contain no FAV devices, in such cases IAVs are the only entities which may control other devices. Although not equipped with a runtime environment for uploaded DCMs, an IAV may be shipped with a set of *embedded* DCMs. Embedded DCMs can be implemented as native applications on the IAV device and can use native interfaces to access the IAV's display and other resources. Embedded DCMs, like DCMs in general, appear in the Registry provided by the HAVi Architecture and can be accessed from other devices on the home network by sending messages over the Messaging System. Embedded DCMs, like DCMs in general, may support the DDI protocol and so may participate in providing a user interface for the controlled device.

2.7 Interoperability in the HAVi Architecture

The first and foremost goal of the HAVi Architecture is to support interoperability between AV equipment. This includes existing equipment and future equipment. Because of the need to support existing devices, and because of product cost considerations, the HAVi Architecture supports two levels of interoperability. These are referred to as Level 1 and Level 2 respectively.

The flexibility of choosing different levels of interoperability is essential in allowing vendors the freedom to design and build devices at all points on the cost/capability spectrum.

2.7.1 Level 1 Interoperability

Level 1 interoperability addresses the general need to allow existing devices to communicate. To achieve this, Level 1 interoperability defines and uses:

▨ a generic set of control messages (commands) that enable one device to talk to another device and

▨ a set of event messages that it should reasonably expect from the device.

To support this approach a basic set of mechanisms are required.

▨ *Device discovery*: each device in the home network needs a well-defined method that allows it to advertise its capabilities to others. The approach the HAVi Architecture has adopted is to utilize SDD data, required on all FAV, IAV and BAV devices. SDD data contains information about the device which can be accessed by other devices. The SDD data contains, as a minimum, enough information to allow instantiation of an embedded DCM. This results in registration of device capabilities with the HAVi Registry, allowing applications to infer the basic set of command messages that can be sent to the device.

▨ *Communication*: once an application has determined the capabilities of another device, then it needs to be able to access those capabilities. To achieve this requires a general communication facility allowing applications to issue requests to devices. This service is provided by the HAVi Messaging Systems and DCMs. The application sends HAVi messages to DCMs, the DCM then engages in proprietary communication with the device.

▨ *HAVi message set*: the last mechanism required to support Level 1 interoperability is a well defined set of messages that must be supported by all devices of a particular class. This ensures that a device can work with existing as well as future devices, irrespective of the manufacturer. The HAVi message set includes those messages used for the DDI protocol and so allows DCMs (and applications) to construct a UI on display-capable IAVs and FAVs.

These three basic mechanisms support a minimal level of interoperability. Since any application can query the Registry, any application can determine the message set supported by any DCM. Since any application has access to the Messaging System, any application can interact with the DCM of any device.

2.7.2 Level 2 Interoperability

As described in the previous section, Level 1 interoperability ensures that devices can interoperate at a basic level of functionality. However, a more extensible mechanism is also needed to allow a device to communicate to other devices any additional functionality not present in embedded DCMs. For example, embedded DCMs may not support all features of existing products and are unlikely to support future product categories. Level 2 interoperability provides this mechanism.

To support non-standard features of existing products and to support future products, the HAVi Architecture allows uploaded DCMs as an alternative to embedded DCMs. The uploaded DCM may replace an existing DCM on FAV devices. The HAVi Architecture makes no statement about the source of the uploaded DCM, but a likely technique is to place the uploaded DCM in the SDD data of the BAV device, and upload from the BAV to the FAV device when the BAV is attached to the home network. Because the HAVi Architecture is vendor neutral, it is necessary that the

Modules, if an IAV or FAV has such a feature then the vendor is responsible to assure only Application Modules obtained from secure sources are given the trusted level, with a vendor-dependent verification mechanism.

As for havlets, they are extracted from DCMs or Application Modules through a public method. Thus, a havlet code unit shall be verified whether it is correctly signed in a HAVi-compliant manner before it is installed on an FAV. If the verification fails, the FAV shall not assign the trusted level to the havlet.

It is vendor-dependent whether or not an FAV or IAV allows to install Application Modules or havlets which failed in the security verification. However, even when installation of such untrusted software elements is allowed, it is recommended that such an FAV or IAV has a proprietary mechanism to assure that such an installation is only done with the user's responsibility.

2.9.2 Signature Verification

The procedure of security verification of code resources used on IAVs is vendor dependent. Also, even on FAVs, the procedure of security verification for Application Modules is vendor dependent. However the digital signature algorithm and certification procedures are specified by HAVi, for the verification of uploadable DCM code units and havlet code units.

The procedure of signature verification on FAVs is as follows:

* Uploadable DCMs, uploadable Application Modules and all havlets are obtained from "code units" – these are JAR files.
* As for uploadable DCMs, the JAR file shall be signed in the HAVi-compliant manner, and the signature is verified when the file is loaded into the Java runtime. If there is no signature, or if signature verification fails, the DCM shall not be installed.
* As for havlets, the JAR file may be signed in the HAVi-compliant manner. In that case the signature is verified when the file is loaded into the Java runtime. If signature verification succeeds, then all classes defined in the file are trusted. If there is no signature, or if signature verification fails, all classes defined in the file are untrusted.
* As for Application Modules, the JAR file may have any information for vendor-dependent verification. If the verification succeeds, then all classes defined in the file are trusted. Otherwise, including the case an FAV is not aware of the verification mechanism, all classes defined in the file are untrusted.
* Only havlets and Application Modules created from trusted classes are trusted.

The digital signature algorithm used for uploadable DCMs and havlets on FAVs, and associated key management infrastructure, are specified in section 3.10.

3 Software Element Descriptions

3.1 Communication Media Manager

The Communication Media Manager (CMM) is a network dependent entity in the HAVi Architecture. It interfaces with the underlying communication media to provide services to other HAVi components or application programs residing on the same device as the CMM. Each physical communication medium has its own CMM to serve the above purpose. This section concentrates only on the CMM for the 1394 bus.

Two types of services are provided by the CMM. One is to provide a transport mechanism to send requests to and receive indications from remote devices. The other is to abstract the network activities and present information to the HAVi system. The 1394 bus is a dynamically configurable network. After each bus reset, a device may have a completely different physical ID than it had before. If a HAVi component or an application has been communicating with a device in the network, it may want to continue the communication after a bus reset, though the device may have a different physical ID. To identify a device uniquely regardless of frequent bus resets, the Global Unique ID (GUID) is used by CMM and other HAVi entities. A GUID is a 64 bit number that is composed of 24 bits of node-vendor ID and a 40 bit number assigned by the vendor (these are described in references [2] and [3]). While a device's physical ID may change constantly, its GUID is permanent. The CMM makes device GUID information available for its clients.

One of the advanced features the 1394 bus provides to the HAVi system is its support for dynamic device actions such as hot plugging and unplugging. To fully support this up to the user level, HAVi system components or applications need to be aware of these network changes. The CMM works with the Event Manager to detect and announce such dynamic changes in network configuration. Since any topology change within the 1394 bus will cause a bus reset to occur, the CMM can detect topology changes and post an event to the Event Manager about these changes along with associated information.

Trusted software elements are allowed to use their local CMM to initiate communication with other devices on the 1394 network, using the Cmm1394::Write, Cmm1394::Read and Cmm1394::Lock APIs defined in section 5.2.2. These APIs would typically be used by DCMs and FCMs to control remote 1394 devices, i.e. BAV and LAV devices. Communication between IAVs and FAVs, devices which support HAVi messaging, is typically accomplished using the HAVi defined software elements or the Messaging System directly.

For a software element to accept 1394-level communication initiated by a remote device it must first enroll for indications by calling the Cmm1394::EnrollIndication API. This API enables the software element to be notified of any write, read or lock operation from a specific device in a specific address range. This notification is performed by the Cmm1394Indication client API. The software element can drop previously enrolled indications by calling the Cmm1394::DropIndication API.

The CMM is intended to allow maximum flexibility so that DCMs and FCMs can individually control remote 1394 devices with a wide variety of functions and protocols. On a single IAV or FAV there could be a number of different DCMs and FCMs, each one controlling different remote 1394 devices using various 1394 address ranges. Note that if different 1394 remote devices perform a read of a given address range, they may receive different results, since each read request may be processed by an independent DCM or FCM. Also note that the response to such a read request (or lock request) is provided by the responseData argument of the Cmm1394Indication message,

rather than by a simple memory mapping.

3.2 Messaging System

3.2.1 Description

The Messaging System provides HAVi software elements with communication facilities. It is independent of the network and transport layers. A Messaging System is embedded in all FAV and IAV devices. The Messaging System of a device is in charge of allocating identifiers (SEIDs) for the software elements of that device. These identifiers are first used by the software elements to register. They are then used by the software elements to identify each other within the home network: when a software element (A) wants to send messages to another software element (B) it has to use the software element identifier of B when invoking the Messaging System API.

The Messaging System is composed of the message layer and the *Transport Adaptation Module* (TAM) which provides a basic communication API that is medium dependent. The TAM uses the services of 1394 and IEC 61883 protocol layers to send and receive data on the network. It gathers features that are medium dependent. When the medium changes, the TAM has to change as well. The TAM is described in section 3.2.2.

3.2.1.1 Software Element Identifier Allocation

A software element identifier (SEID) is allocated by the Messaging System when requested by a software element. A software element must obtain a SEID if it wants to be registered on the home network, or if it wants to communicate (via HAVi messages) with other software elements.

The software element identifier is 10 bytes long.

Syntax	Number of bits	Identifier
SEID() {		
GUID	64	uimsbf
swHandle	16	uimsbf
}		

Figure 6. SEID Representation

GUID identifies a device within the home network. It is the 1394 EUI64 value that is available in the ROM of 1394 devices.

swHandle identifies a software element within one device. swHandle allocation is described in section 3.2.1.1.1.

SEID, being the concatenation of GUID and swHandle, identifies therefore a software element within the home network.

From the software element's point of view, the identifier is an atomic identifier that is 10 bytes long. Only the HAVi Messaging System and other HAVi system components are aware of the internal structure of software element identifiers.

3.2.1.1.1　　　Software Element Handle Allocation

swHandle is allocated by the Messaging System of a device when a software element requests a software element identifier. The Messaging System of a device is in charge of allocating unique handles to the local software elements. *It is recommended that the Messaging System avoids reuse of the handles if possible so as to avoid potential conflict* (see section 3.2.1.2).

3.2.1.1.2　　　Well-known Software Element Handles

There is a need to have well-known software element handles for system components. The handle values from 0x0 to 0x00ff are reserved for this purpose. These well-known software element handles are listed in Annex 11.4. To reach a system component the requester must know the SEID of the component. Usually it will use the Registry service to find SEIDs. Therefore the requester has to reach the local Registry API and, consequently has to know the local Registry SEID. To avoid such circularity, the Messaging System API provides a method to find the SEIDs of system components.

3.2.1.1.3　　　Trusted and Untrusted Software Element Handles

HAVi defines a set of APIs provided by system components and DCM/FCMs. Some of the API methods are protected in the sense that only a subset of software elements are authorized to access the method. Consequently software element are classified into two categories:

- trusted software elements (can access all HAVi APIs)

- untrusted software elements (can access only untrusted APIs)

Each API described in chapters 5 and 6 gives, for each API method, whether it can be used by trusted and untrusted software elements or just trusted software elements.

To identify whether a software element is trusted or not, two software element handle ranges are defined :

0x0000 to 0x7fff : trusted software elements

0x8000 to 0xffff : untrusted software elements

The well-known handles (i.e., system component handles – see Annex 11.4) are all in the trusted range. The Messaging System delivers a trusted SEID or an untrusted SEID to a requester according to the access level of this requester (see section 2.9).

3.2.1.2　　Message Transfer Service

The Messaging System provides a connectionless data transfer service. Before being able to exchange any kind of information, a software element has to: 1) obtain a software element identifier (SEID), and 2) indicate a call back function to the Messaging System for receiving messages. This step is done via the MsgOpen function.

Once a software element has obtained an identifier, and should it want to send messages to another software element (known through its software element identifier), it must use the message transfer services of the Messaging System.

3.2.1.2.1   *Message Transfer Supervision*

When a software element wants to send data to another software element it may establish a *supervision* of that software element via the MsgWatchOn function. The purpose of having a supervision of a software element is to be told when a software element leaves the network. As long as a supervision is established, the Messaging System is in charge of detecting when a device (on which it has established a supervision) disappears from the network. If this occurs it shall close all its supervisions to the disappeared device by invoking the call back of the source software elements.

A supervision is always unidirectional: if the destination software element needs to be informed of the loss of the originating software element, then it must establish its own supervision of that software element.

After a supervision has been requested, the Messaging System monitors the existence of the "watched" object using the Msg::Ping facility described in section 5.3.3. The Messaging System also subscribes to events that allows it to detect: when the device hosting the destination software element goes down, when the destination software element leaves the network, when the remote device is initialized (reset), and when the remote HAVi system has entered an anomalous state..

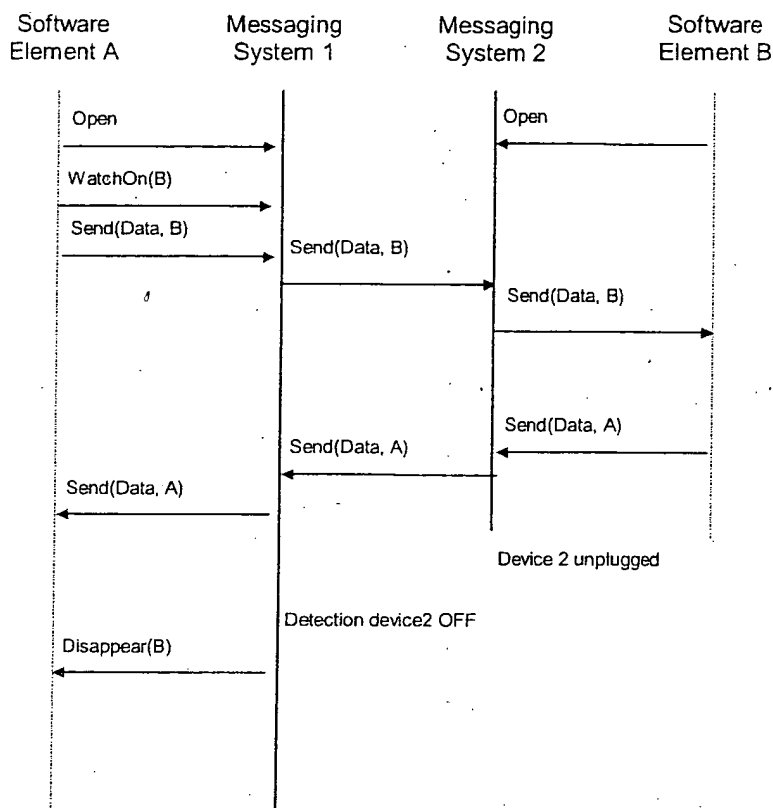Figure 7.   **Example of Message Transfer Supervision**

3.2.1.2.2   *Message Transfer Modes*

Data are not protected at the message level (i.e., there is no Messaging System CRC). However it is assumed that the lower layers provide an error detection service (via a CRC) as well as a good level of transmission reliability (see section 3.2.2). Therefore there are no error recovery mechanisms at

the message level (however the TAM provides an error recovery process according to 1394).

The Messaging System provides two modes to transmit a message: *simple mode* and *reliable mode*. The Messaging System API primitives MsgSendReliable, MsgSendRequest and MsgSendRequestSync use reliable mode, while MsgSendSimple uses simple mode and MsgSendResponse takes a mode parameter (see section 5.3.3).

Simple mode is very basic: no control is performed by the Messaging System. The message is sent on the network and that is all. This mode may be used when a software element wants to send a response to a request. The Messaging System does not check whether its response is received or not.

Reliable mode is more complicated and expects the destination device to acknowledge the message. The reliable mode is described through the following example: when a software element (A) running on a device D1 wants to send a reliable message to another software element (B) running on device D2, it invokes a Messaging System API with B's SEID as parameter. The Messaging System of D1 sends a msg_reliable message (see 3.2.1.2.4) to the Messaging System of D2. The Messaging System of D2 checks then whether it knows a call back for B. If yes, it invokes it. If the call back successfully returns (i.e., no error), it sends back a msg_reliable_ack to D1 (as an acknowledgement). If either no call back is available, or the call back returns with an error, D2's Messaging System sends back a msg_reliable_noack message which indicates an error. When the Messaging System of D1 receives the msg_reliable_ack (or msg_reliable_noack) message, the Messaging System API function invoked by A returns. Note that at the originating side, the calling software element is blocked until it gets the acknowledgement. To avoid blocking a software element indefinitely an acknowledgement timeout is used. Its value shall be 30 seconds. Expiry of the timer indicates to the Messaging System a major problem either locally or on the target. The timer is started at the end of transmission of the last TAM package of the message. In case device D1 detects that device D2 has disappeared before the msg_reliable_ack has been received, the Messaging System of D1 shall immediately return from MsgSendReliable with the error code Msg::EACK. In this situation no MsgTimeout event is generated.

3.2.1.2.3   Acknowledgements

The general message structure includes an 8-bit field called the *message number*.

The Messaging System maintains a message number counter for each source software element. When the Messaging System issues a request (reliable or not), the counter is incremented (modulo 256) and the new value is sent within the message. In case of a msg_reliable message, the Messaging System at the destination node will send back the same counter value within the msg_reliable_ack (or msg_reliable_noack) message.

The Messaging System checks each incoming msg_reliable_ack or msg_reliable_noack. If no reliable message of the destination software element with the same message number is pending, the incoming ack (or noack) is simply discarded.

In the case the response is received prior to the msg_reliable_ack of a request, the Messaging System may treat the response as a msg_reliable_ack. This may recover from a lost msg_reliable_ack in the case where the response is received before the acknowledgement timeout.

The Messaging System that resides on the same node as the software element receiving the request message shall send msg_reliable_ack and response message in order. Thus, the software element which receives a callback invocation has to immediately return from the callback.

Software Element A	Messaging System 1	Messaging System 2	Software Element B

msg_send_reliable(B) Invocation

msg_ reliable (B)

CallBack(req) Invocation

CallBack(req) Return

msg_ reliable _ack(A)

msg_send_reliable(B) Return : ok
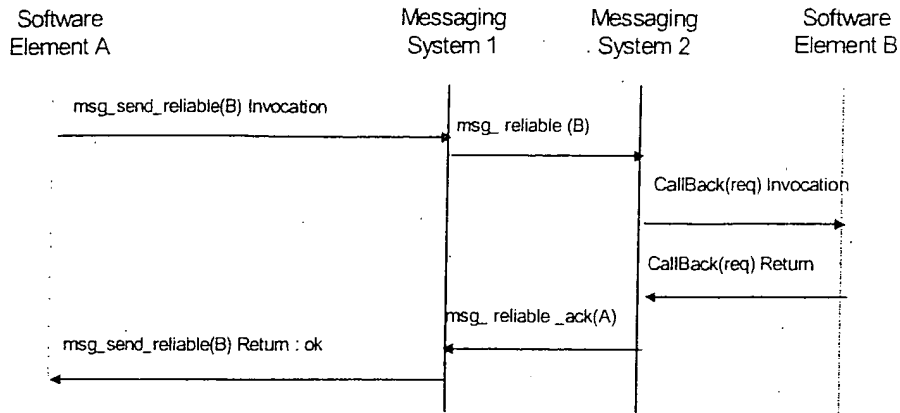
Figure 8. Typical Reliable Message Sequences

Software Element A	Messaging System 1	Messaging System 2	Software Element B

msg_send_reliable(B) Invocation

msg_ reliable(B)

CallBack(req) Invocation

CallBack(req) Return

Reliable ack lost

msg_send_reliable(B) Return : timeout
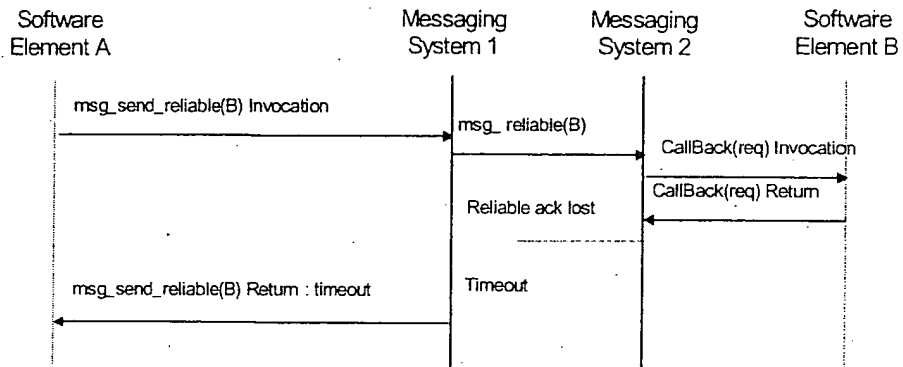
Timeout

Figure 9. Reliable Messaging Failing Due to Timer Expiration

3.2.1.2.4   *General Message Format*

The message format shown in Figure 10 describes a message as it is sent by a Messaging System to another Messaging System.

Syntax	Number of bits	Identifier
message () {		
DestSEID	80	uimsbf
SourceSEID	80	uimsbf
ProtocolType	8	uismbf
MessageType	8	uimsbf
MessageNumber	8	uimsbf
reserved	8	uimsbf
MessageLength	32	uimsbf
for(j=0; j<MessageLength;j++){		
MessageBody[j]	8	uimsbf
}		
}		

Figure 10. General Message Format

DestSEID (10 bytes) is the identifier of the software element to which the message is to be sent.

SourceSEID (10 bytes) is the identifier of the software element that generated the message.

Note – When the message is a msg_reliable_ack or msg_reliable_noack, the DestSEID field will hold the SourceSEID of the related msg_reliable message.

ProtocolType (1 byte) is the format that MessageBody content must adhere to. The values from 0x00 to 0x7f are reserved for HAVi. The values from 0x80 to 0xff are free for private use. HAVi defines one particular protocol based on request and response message exchanges, providing applications with facilities to invoke operations on a given software element (request), and/or to return the result of an operation to the software element that requested it (response). The ProtocolType parameter in the general message format corresponding to the HAVi request/response mechanism has the value 0x00. For this value of ProtocolType, if MessageType (see below) is msg_simple or msg_reliable then the message body format must be as specified in section 3.2.3.2.

MessageType (1 byte) indicates the message type:

  ※ One type is defined for the simple mode : the msg_simple message
  ※ Three types are defined for the reliable mode: the msg_reliable message (carrying the request), the msg_reliable_ack (telling the request has been delivered), and the msg_reliable_noack (telling the request delivery failed).

MessageNumber (1 byte) is the message number. Its value is incremented according to the rules described in section 3.2.1.2.3.

reserved (1 byte), this field shall be set to zero.

MessageLength (4 bytes) is the length of the MessageBody. There is no payload for the msg_reliable_ack, in which case both the MessageLength and MessageBody fields are not

present.

MessageBody is the message data.

Table 2. Message Type Values

MessageType	Value
msg_simple	0x01
msg_reliable	0x02
msg_reliable_ack	0x03
msg_reliable_noack	0x04

3.2.1.2.5   Ack Message Format

The msg_reliable_ack message follows the general message format. It has no MessageLength and no MessageBody. The layout of an msg_reliable_ack message is indicated schematically below:

byte 0	byte 1	byte 2	byte 3	byte 4	...	byte 13
0011 rrrr	nnnn nnff	0Rnn nnnn	0000 0000	dddd dddd	...	dddd dddd
FCPHdr	TAMHdr		reserved	Destination SEID		

byte 14	...	byte 23	byte 24	byte 25	byte 26	byte 27
ssss ssss	...	ssss ssss	PPPP PPPP	0000 0011	NNNN NNNN	0000 0000
Source SEID			ProtType	MsgType	MsgNo	reserved

Destination SEID in the above is the SEID of the software element that sent the corresponding msg_reliable message. Source SEID is the SEID of the software element that received the msg_reliable message. The value of protocol type shall be the same as the value of the corresponding msg_reliable message.

3.2.1.2.6   Noack Message Format

The msg_reliable_noack message follows the general message format. The MessageBody contains one byte which may take the values shown below:

Table 3.   msg_reliable_noack Message Body Values

Name	Value
SYSTEM OVERFLOW	0x01
UNKNOWN TARGET OBJECT	0x02
TARGET REJECT	0x03

SYSTEM OVERFLOW – memory allocation for the incoming message failed.

UNKNOWN TARGET OBJECT – the Messaging System cannot find a callback for the destination SEID embedded in the incoming message, or cannot deliver the response message for the synchronous function call to the caller software element.

TARGET REJECT – the return value of the callback is not equal to SUCCESS.

The layout of a msg_reliable_noack message is indicated schematically below:

byte 0	byte 1	byte 2	byte 3	byte 4		byte 13
0011 rrrr	nnnn nnff	0Rnn nnnn	0000 0000	dddd dddd	...	dddd dddd
FCPHdr	TAMHdr		reserved	Destination SEID		

byte 14		byte 23	byte 24	byte 25	byte 26	byte 27
ssss ssss	...	ssss ssss	PPPP PPPP	0000 0100	NNNN NNNN	0000 0000
Source SEID			ProtType	MsgType	MsgNo	reserved

byte 28	byte 29	byte 30	byte 31	byte 0
0000 0000	0000 0000	0000 0000	0000 0001	eeee eeee
Message Length				Message Body

Destination SEID in the above is the SEID of the software element that sent the corresponding msg_reliable message. Source SEID is the SEID of the software element that received the msg_reliable message. The value of protocol type shall be the same as the value of the corresponding msg_reliable message.

3.2.1.2.7 HAVi Message Version

The HAVi message version supported by a device is in its **HAVi_Message_Version** SDD field. Before the first exchange between two Messaging System modules, the initiator obtains the message version number from the receiver device. Then the initiator will use the message format defined in the highest HAVi specification shared by both Messaging Systems.

3.2.1.2.8 Outstanding Message

Outstanding message means that a message is in the midst of a message transfer (for simple mode) or waiting for a corresponding acknowledgement (for reliable mode), i.e. "outstanding" starts when the valid message number is given to the message and ends when the message number is retrieved. If the number of outstanding messages exceeds the outstanding message limit for each Software element, the messaging system shall return the error code EBUSY. A messaging system implementation may send multiple messages in parallel as long as it does not exceed the outstanding message limit for the same source SEID. Outstanding message limit is implementation dependent and its maximum value is 256.

3.2.2 Transport Adaptation Module (TAM)

3.2.2.1 Service Description

The part of Messaging System which is medium dependent is called the TAM. The TAM manages message fragmentation, and the message ordering and error recovery process if needed. For these purposes it defines a packet format and a protocol.

The TAM sends and receives data on the IEEE 1394 bus in the range of the FCP command register. The TAM is only notified of indications within this range which have the HAVi CTS code. A Messaging System which has enrolled for indications in the FCP address range does not influence or block other software elements from using the CMM to enroll for indications in that address range.

3.2.2.2   Fragmentation

Fragmentation is performed according to the network capabilities. For IEC 61883/1394 this service does not exist. Therefore a fragmentation service is used in the TAM to perform fragmentation on the messages generated by the Messaging System.

The TAM packet header has a part dedicated to fragmentation. If fragmentation is used then the TAM has to serialize message transmission for a particular destination node (i.e., all TAM packets for the current message will be sent before processing the next pending message for the same destination node). The TAM can parallelize message transmissions for different destination nodes.

3.2.2.3   Message Ordering

Message ordering is performed according to the network technology. For IEC 61883/1394 this service does not exist. Therefore an ordering service is performed by the TAM.

A continuity counter present within the TAM packet header is incremented (modulo 64) before each TAM packet transmission. On the receiving side, the continuity counter is processed in relation to the source GUID extracted from the 1394 packet header (and to point to the sender context).

3.2.2.4   Mapping of TAM onto the 1394 Transaction Layer

TAM packet write requests are mapped into 1394 transaction write requests (possibly via the CMM). When writing several TAM packets to a single destination a TAM shall not parallelize transactions: before generating a transaction write request to a destination it has to wait for completion of the previous transaction (to that destination).

3.2.2.4.1   IEC 61883 FCP Packet

TAM data packets are sent or received according to the IEC 61883 FCP protocol and format (see [4]).
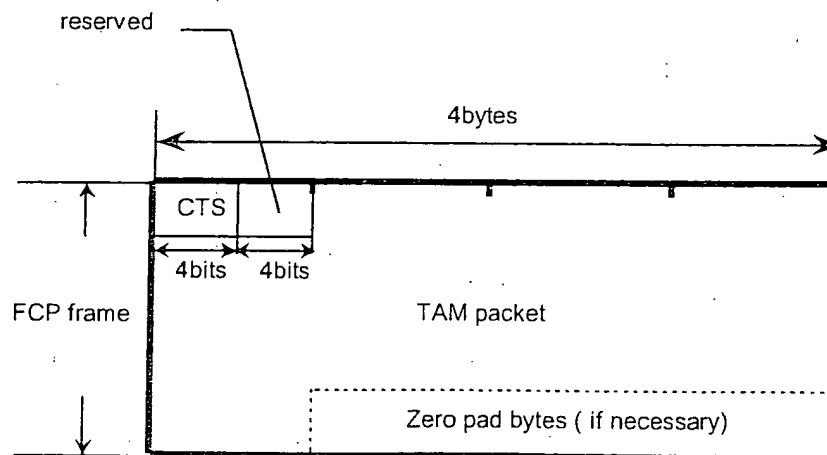

Figure 11. IEC 61883 FCP Packet Structure

A new CTS code is reserved for HAVi message transport. The following four bits after the CTS code are also reserved by HAVi; they shall be set to 0. Note that Figure 11 shows a generic IEC

34

61883 FCP packet that is padded with zero bytes to make the packet size a multiple of 4 bytes. In compliance with 1394-1995 the TAM layer shall indicate in the length field of the 1394 header only its own payload. The padding bytes, if any, are not included in this length.

CTS code				CTS
b7	b6	b5	b4	
0	0	0	0	AV/C
0	0	0	1	CAL
0	0	1	0	EHS
0	0	1	1	HAVi
⟨				⟨
1	1	0	1	(Reserved)
1	1	1	0	Vendor Unique
1	1	1	1	Extended CTS

Figure 12. IEC 61883 CTS Codes

Concerning the transmission of messages, only the IEC61883 FCP command register is used to store a TAM data packet.

3.2.2.4.2   TAM Data Packet Structure

In case of transmission of a HAVi message, the TAM has to provide a mechanism to manage the fragmentation of messages and to preserve message ordering. For these purposes a TAM packet contains a header which permits the receiver to assemble the message.

Syntax	Number of bits	Identifier
TAM_HaviDataPacket () {		
SequenceNum	6	uimsbf
FragType	2	uimsbf
Reserved	1	uimsbf
RetryFlag	1	uimsbf
OriginalSeqNum	6	uimsbf
Reserved	8	uimsbf
for(j=0; j<FragData size; j++){	8	uimsbf
FragData[j]		
}		
}		

Figure 13. TAM_HaviDataPacket Representation

SequenceNum is used to assemble the various fragments from a message. A TAM sender shall maintain one SequenceNum for each possible TAM receiver. The sender shall continuously increment this number after every fragment is transmitted. The number shall not be reset to zero at the end of transmission of the whole message. Therefore SequenceNum can also be used to

guarantee message ordering.

FragType (Fragment Type) is a 2 bit value as defined in Table 4.

Reserved is a one bit field that shall be set to zero.

RetryFlag is a one bit field that indicates if the packet is a retry or not. If set to one, the packet is a "resent" packet. If set to zero, the packet is "fresh".

OriginalSeqNum is a 6 bit field that indicates the original SequenceNum of the failed original packet. If RetryFlag is zero, this field shall be set to zero.

Reserved is a one byte field that shall be set to zero

FragData is a variable number of bytes and may be null but shall never exceed a maximum limit, which is dependent on the target (the device receiving the TAM packet). This maximum limit derives from the maximum 1394 data payload size the target node accepts. The "max_rec" field of the configuration ROM Bus_Info_Block defines this value. Thus the maximum size of the FragData field is computed as follows:

maximum FragData size = "max_rec"– TAM_header_size – FCP_header_size

where TAM_header_size = 3 and the FCP_header_size = 1.

A compliant IAV or FAV must be able to accept block write requests of at least 8 bytes; TAM data packets shall never exceed 512 bytes. For the "max_rec" Bus_Info_Block field, a value of 0 indicates "Not Specified", which is interpreted as being able to handle at least 512 bytes. A value of 1 indicates a capacity of 4 bytes, which is not allowed for a compliant IAV of FAV implementation. Values greater than 8 indicate a capacity of at least 512 bytes.

Table 4.   TAM Fragment Type Values

FragType	Value	Meaning
SFP	0x00	Simple Fragment Packet
BOP	0x01	Begin Of message Packet
COP	0x02	Continuation Of message Packet
EOP	0x03	End of message Packet

3.2.2.5   Reliable TAM Packet Transmission

If a bus reset or a transmission error occurs during a 1394 transaction, the transaction is aborted and 1394 packets may be lost. To recover from lost packets the TAM shall implement the following mechanism:

A TAM sender increases the continuity counter of the TAM packet header (see sections 3.2.2.3 and 3.2.2.4.2). If the transaction fails (reception of a transaction_conf with an error code), the TAM shall repeat the packet (without increasing the continuity number). The number of consecutive retries cannot exceed three. In case of failure the Messaging System returns a Msg::ESEND error (see section 5.3.3). After a Msg::ESEND error occurs, for the next packet sent to the same receiver, the sender shall increase the TAM sequence number.

Note that a TAM sender assumes the transaction succeeded as soon as it receives a transaction_conf (i.e., an acknowledgement from the other side). Finally a TAM sender has to wait for a transaction_conf from one destination before generating another transaction to that destination (increasing the continuity number).

If the TAM sender detects a bus reset before receiving a transaction_conf from the 1394 transaction layer, the sender has to repeat the complete message after sending a synchronization

packet as specified in 3.2.2.6. In case of an interrupted BOP, COP or EOP packet, this requires re-sending packets beginning with the BOP packet. In case of re-sending , TAM sender shall set RetryFlag to one and OriginalSeqNum to the corresponding sequence number of the first resent SFP packet or to the sequence number of the first resent BOP, COP or EOP packet. TAM receiver shall check if the incoming packet is a "resent "or a "fresh" packet with RetryFlag. If the packet is a "resent" packet, also check if the packet is already received or not with OriginalSeqNum. If the packet is already received, the packet shall be discarded.

A TAM receiver checks the continuity number for packet reordering and re-assembly. A 1394 receiver shall not acknowledge a transaction (either in a unified[1] or split transaction[2]) before being sure the 1394 packet has successfully been received. Once the transaction has been acknowledged the receiver ensures the packet will not be discarded by any bus reset. (If the TAM uses unified transactions that are acknowledged by hardware and not by the TAM itself, it shall ensure that once acknowledged a packet is assumed to be received by the TAM and that it will not be discarded by a bus reset.) If a TAM receives twice a packet with the same continuity number, it shall discard one.

A receiver shall accept a packet with any sequence number that differs from the sequence number of the previously received packet from the same sender. If an SFP or BOP packet is received, a previous sequence of BOP and/or COP packets that has not been concluded by an EOP packet shall be discarded.

3.2.2.6   *TAM Sequence Number Synchronization*

The first TAM packet sent from node A to node B after A is powered up, is reset, or has detected a 1394 bus reset event, shall be an SFP packet with a zero payload (a "synchronization packet"). This SFP packet shall not be passed to a software element, but is used only to synchronize the TAM sequence numbers of sender (A) and receiver (B). If the synchronization packet carries sequence number n, then the next packet sent from A to B shall have number $n+1$ (*modulo 64*). Future packets sent from A to B shall have normally increasing sequence numbers. A receiver shall accept a synchronization packet at anytime.

Consider the situation where A and B are disconnected and reconnected very quickly. A may be a "slow" device. Thus it never detects the disappearance of B. B may be a fast device. Thus it can detect the disappearance and the re-appearance of A. The sequence number B expects from A will be unknown, so B shall expect a synchronization packet from A.

3.2.3 Mapping of Function Calls into Messages

IDL is the basis for mapping software element public APIs into messages.

3.2.3.1   *Mapping of an IDL Interface into the Messaging System API*

The APIs of software elements (system services, FCMs and DCMs) are specified as a set of operations. When a software element wants to invoke a function of another software element, it maps the function call into a message and it sends the message to that software element.

---

[1] In the case of a unified transaction, the acknowledgement (that ends the transaction) is the acknowledgement of the unified write.

[2] In the case of a split transaction, the acknowledgement (that ends the transaction) is the response subaction of the remote TAM.

Afterwards, the called software element may want to send a response (if the IDL operation contains a return value, inout or out parameters). To do so it sends to the calling software element.

Some general rules are needed for mapping functions into messages. Such rules are described in the following sections.

As a starting point, consider the following IDL operation:

```
IDL_Type IDL_operation(
        param_attribute1 param_type1, param_name1,
        param_attribute2 param_type2, param_name2,
        param_attribute3 param_type3, param_name3,
        param_attribute4 param_type4, param_name4 ...
```

3.2.3.2   Mapping of Function Calls into Messages

When a software element A invokes an operation (IDL_operation) on a software element B, the operation is mapped to a message. The Messaging System shall use the following format:

Syntax	Number of bits	Identifier
msg_function_call () {		
DestSEID	80	uimsbf
SourceSEID	80	uimsbf
ProtocolType	8	uimsbf
MessageType	8	uimsbf
MessageNumber	8	uimsbf
reserved	8	uimsbf
MessageLength	32	uimsbf
OperationCode	24	uimsbf
ControlFlags	8	uimsbf
TransactionId	32	uimsbf
for (j=0; j<param_number;j++){		
if ((param_attributej == IN) ||		
(param_attributej == INOUT)) {		
param_valuej	sizej	uimsbf
}		
}		
}		

Figure 14. Function Call Mapping to a Message

DestSEID (10 bytes) is the identifier of the software element to which the message is to be sent. It is the software element that shall execute the function.

SourceSEID (10 bytes) is the identifier of the software element that generates the message. It is the software element that calls the function.

ProtocolType (1 byte) is the format that MessageBody content must adhere to. The possible values for this field are listed in Annex 11.1. HAVi defines one particular protocol, HAVi_RMI, based on request and response message exchanges. This protocol allows one software element to invoke an operation on another (request), and it allows the result of the operation to be returned (response).

MessageType as described in Table 2 (shall be msg_reliable).

MessageNumber (1 byte) is the message number. Its value is incremented according to the rules described in section 3.2.1.2.3.

reserved (1 byte) shall be set to zero.

MessageLength is the number of following bytes. This length includes the OperationCode, ControlFlags and TransactionId lengths and the parameter loop length.

OperationCode is defined for each software element API. HAVi operation codes are listed in Annex 11.6.

ControlFlags is an 8-bit field where only the lowest bit is currently used. This bit is called the RequestFlag and takes the value 0. It indicates that the message is carrying a call operation request.

TransactionId is provided by the requester. The receiver has to put this TransactionId in its response message. It allows the requester to match a response with a request in case of multiple requests to the same object.

param_value$j$ is the value of the jth IN or INOUT parameter of the operation specified by OperationCode. The parameter size, size$j$, includes any padding added by the conversion to CDR.

The layout of a message carrying a function call is indicated schematically below:

byte 0	byte 1	byte 2	byte 3	byte 4	...	byte 13
0011 rrrr	nnnn nnff	0Rnn nnnn	0000 0000	dddd dddd	...	dddd dddd
FCPHdr	TAMHdr		reserved	Destination SEID		

byte 14	...	byte 23	byte 24	byte 25	byte 26	byte 27
ssss ssss	...	ssss ssss	0000 0000	0000 0010	NNNN NNNN	0000 0000
Source SEID			ProtType	MsgType	MsgNo	reserved

byte 28	byte 29	byte 30	byte 31
LLLL LLLL	LLLL LLLL	LLLL LLLL	LLLL LLLL
Message Length			

byte 32	byte 33	byte 34	byte 35
oooo oooo	oooo oooo	oooo oooo	cccc ccc0
OperationCode			CtrlFlag

byte 36	byte 37	byte 38	byte 39
TTTT TTTT	TTTT TTTT	TTTT TTTT	TTTT TTTT
Transaction Id			

3.2.3.3 Mapping of Function Returns into Messages

When a software element A invokes a function on another software element B using a request

39

message, and if the function gathers INOUT, OUT or a return code, the called software element shall map the output of the function to a message. The Messaging System shall use the following format:

Syntax	Number of bits	Identifier
msg_function_response () {		
DestSEID	80	uimsbf
SourceSEID	80	uimsbf
ProtocolType	8	uimsbf
MessageType	8	uimsbf
MessageNumber	8	uimsbf
reserved	8	uimsbf
MessageLength	32	uimsbf
OperationCode	24	uimsbf
ControlFlags	8	uimsbf
TransactionId	32	uimsbf
ReturnApiCode	16	uismbf
ReturnErrCode	16	uimsbf
reserved	32	uimsbf
for (j=0; j<param_number;j++){		
if ((param_attributej == OUT) ||		
(param_attributej == INOUT)) {		
param_valuej	sizej	uimsbf
}		
}		
}		

Figure 15. Function Return Mapping to a Message

DestSEID (10 bytes) is the identifier of the software element to which the message is to be sent. It is the software element that calls the function and that will get the function output.

SourceSEID (10 bytes) is the identifier of the software element that generates the message. It is the software element that has executed the function that sends the output.

ProtocolType (1 byte) is the format that MessageBody content must adhere to. The possible values for this field are listed in Annex 11.1. HAVi defines one particular protocol, HAVi_RMI, based on request and response message exchanges. This protocol allows one software element to invoke an operation on another (request), and it allows the result of the operation to be returned (response).

MessageType as described in Table 2.

MessageNumber (1 byte) is the message number. Its value is incremented according to the rules described in section 3.2.1.2.3.

reserved (1 byte) shall be set to zero.

MessageLength is the number of following bytes.

OperationCode is defined for each software element API. HAVi operation codes are listed in Annex 11.6.

ControlFlags is an 8-bit field where only the lowest bit is currently used. This bit is called the ResponseFlag and takes the value 1. It indicates that the message is carrying an operation response.

TransactionId is provided by the requester. The receiver inserts this TransactionId in its response message. It allows the requester to match a response with a request in case of multiple requests to the same object.

OperationReturnCode contains the return code of the IDL operation.

ReturnApiCode and ReturnErrCode together contain the return code (i.e., Status, see 5.1.2) of the IDL operation.

reserved (4 bytes) shall be set to zero.

param_value$_j$ is the value of the jth OUT or INOUT parameter of the operation specified by OperationCode. The parameter size, size$_j$, includes any padding added by the conversion to CDR.

The layout of a message carrying a function return is indicated schematically below:

byte 0	byte 1	byte 2	byte 3	byte 4	...	byte 13
0011 rrrr	nnnn nnff	0000 0000	0Rnn nnnn	dddd dddd	...	dddd dddd
FCPHdr	TAMHdr		reserved	Destination SEID		

byte 14	...	byte 23	byte 24	byte 25	byte 26	byte 27
ssss ssss	...	ssss ssss	0000 0000	0000 00tt	NNNN NNNN	0000 0000
Source SEID			ProtType	MsgType	MsgNo	reserved

byte 28	byte 29	byte 30	byte 31
LLLL LLLL	LLLL LLLL	LLLL LLLL	LLLL LLLL
Message Length			

byte 32	byte 33	byte 34	byte 35
0000 0000	0000 0000	0000 0000	cccc ccc0
OperationCode			CtrlFlag

byte 36	byte 37	byte 38	byte 39
TTTT TTTT	TTTT TTTT	TTTT TTTT	TTTT TTTT
Transaction Id			

byte 40	byte 41	byte 42	byte 43
AAAA AAAA	AAAA AAAA	EEEE EEEE	EEEE EEEE
ApiCode		ErrCode	

byte 44	byte 45	byte 46	byte 47
0000 0000	0000 0000	0000 0000	0000 0000
reserved			

The field "MsgType" (MessageType) is either 0000 0001 or 0000 0010.

3.2.3.4 Mapping of IDL Types and Parameters to Bitflows

The mapping uses a subset of CDR (Common Data Representation) from GIOP (General Inter Orb Protocol) version 1.1 [5] as the transfer syntax for the parameters. Each parameter is byte aligned according to its natural length, e.g., a parameter of length four bytes should start on a four byte boundary. The following restrictions are used:

* The byte ordering is BIG ENDIAN: MSB (Most Significant Byte) first.

* Since the types of all parameters in HAVi messages can be unambiguously interpreted, CDR type codes are not used.

* Concerning the complex types, repository ID, name and member name are not supported. For example the `tk_struct` is followed by the list of member types of the structure.

* The fixed size for the `wchar` IDL type is 2 bytes according to UNICODE UTF-16 and ISO 10646 UCS-2.

* Padding bytes shall be set to zero.

The starting point of the CDR mapping is the field after `MessageLength` as described in the section on "General Message Format" (see 3.2.1.2.4, Figure 10). Specifically, the area where the CDR mapping is applied is the `MessageBody`.

3.2.3.5 Synchronous Message Transfer Mode

The message passing API will provide a synchronous service allowing a caller to block until a response is received. As shown in the following figure, the caller asks to send a function call through the message passing API. The local Messaging System sends a request message (using reliable mode) and waits for the response or a timeout condition. The remote Messaging System receives the request and passes it to the destination software element. The destination element sends its function response message using a normal send in simple or reliable form. The requester's Messaging System receives the response and transmits it to the requester.

Note that the response message is delivered to the caller software element by completing the `msg_send_sync` invocation. If the mode of the response message is reliable, the requester's Messaging System sends a `msg_reliable_ack` (success to transmit) or `msg_reliable_noack` (failure to transmit) related to the reliable response message, to the remote Messaging System, after the requester's Messaging System receives the response message.
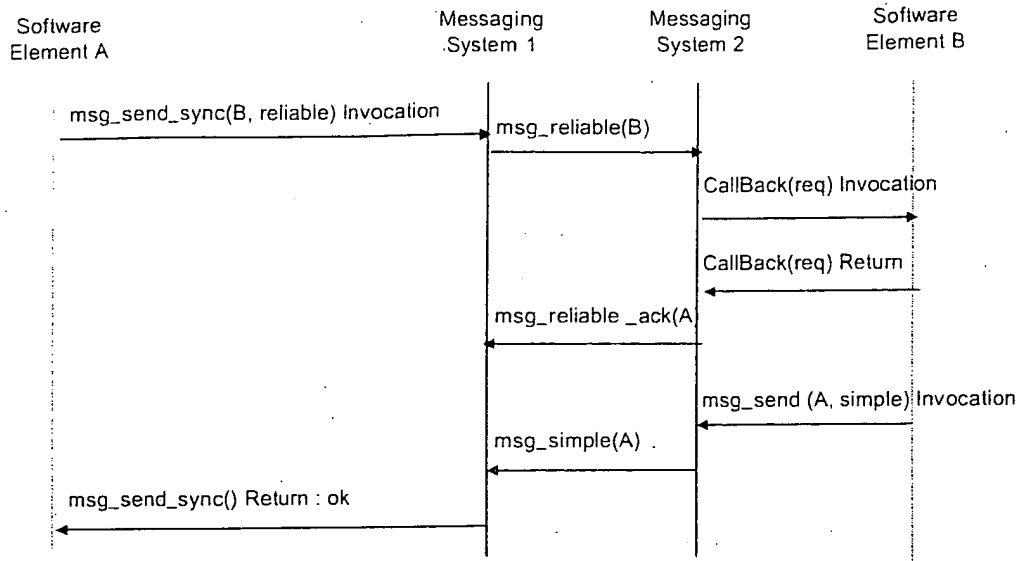
Figure 16. Example of Synchronous Message Transfer

Figure 16 shows an example where the response message is sent in simple mode. It is also possible that the destination element sends the response message in reliable mode.

3.2.4 Implementation Guidelines and Suggestions

3.2.4.1   GUID to phy_id Mapping

The Messaging System uses the GUID (either directly or indirectly through the SEID) to identify a network device in a stable way. It is the responsibility of the Messaging System (and/or the CMM1394) to build and maintain the association table between the GUID and the 1394 phy_id. The building of such a table may be very slow (due to the need to read the GUIDs of all nodes after the detection of the bus reset). To increase the efficiency of this discovery process, the following algorithm, which allows the tracing of the 1394 phy_ids based on the self_id packets, may be used.

After a bus reset the physical addresses of 1394 nodes may change. However each node does not receive information about the addresses other nodes have obtained. HAVi uses a persistent method of node identification based on the 64-bit node unique identifier (GUID). One method of determining the new physical address of a node, given its GUID, is to read the GUID on each node of the network until a match is found. The disadvantage of this method is that it may generate a lot of traffic (in particular if all the nodes use the same technique) and so may be slow.

The aim of the algorithm described here is to re-identify the nodes after a bus reset by using *self_id* packets, and so find the new physical addresses of all nodes. The GUID can then be *directly* read on the node to verify the result.

The main principle is to construct the connection tree of the network before and after the reset, and with this data, to construct a translation table indicating for each node its new number or whether the node has disappeared or is new.

There are two stages:

▓ construction of a connection tree, and

▓ construction of a translation table.

3.2.4.1.1 Connection Tree Construction

The aim of this stage of the algorithm is to build the connection tree of nodes of a 1394 bus with the *self_id* packets.

Note: The 1394 standard assumes a similar procedure to construct the speed map with the topology map.

Information available initially is:

▓ the self-renumeration strategy on a 1394 bus,

▓ for each port of each node, whether there is a node connected to the port and whether it is a child or the parent.

The following paragraph is just a simplified summary of the 1394 self-re-numeration strategy, for more information see section 3.7.3.1 of *IEEE Std 1394-1995 High Performance Serial Bus, 1996-08-30*.

Schematically, after the root election, the node which is root wants to determine its number. When a node wants to know its number, it asks the number of each child going from the lower port to the upper port. The node's number is then the last self_id packet viewed on the bus plus one. This number is then sent via a self_id packet on the bus. For example:
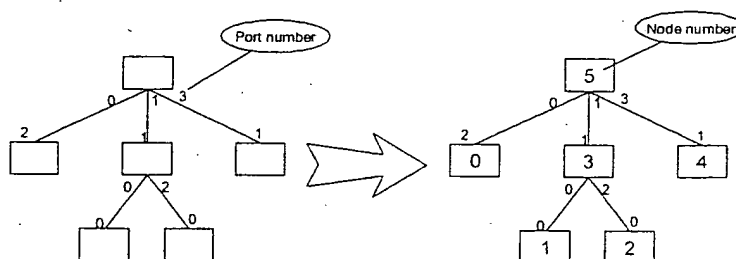

Figure 17. Self-renumeration Strategy

To build a connection tree, perform the following steps using the self_id packets generated during 1394 self-renumeration.

1. Build two node sets: one called *parent-set* containing nodes which have at least one child and one called *child-set* containing nodes which have no child.

2. Take the smallest node in the *parent-set* (let the number be *P*).

3. The number of children of *P* is known, their node numbers are the greatest nodes in the *child-set* with a number less than *P*. Their port is also known (the lowest is on the lowest child port...). Remove the children of *P* from the *child-set*.

4. Move *P* from the *parent-set* to the *child-set*.

5. While the *parent-set* is not empty, go to step 2.

3.2.4.1.2   Example

Using the network in Figure 17, the known information is:

Node	Parent Port	Child Ports	Unconnected Ports
0	2		0,1
1	0		1
2	0		
3	1	0,2	
4	1		0
5		0,1,3	2

Algorithm actions and results:

Description	Parent-set	Child-set	Result
Beginning	3,5	0,1,2,4	
The smallest parent is 3. It has 2 children. The greatest children with number less than 3 are 2 and 1.	5	0,3,4	The node 1 is connected to the port 0 of the node 3.
			The node 2 is connected to the port 2 of the node 3.
The smallest parent is 5. It has 3 children. The greatest children with a number less than 5 are 4, 3 and 0.			The node 0 is connected to the port 0 of the node 5.
			The node 3 is connected to the port 1 of the node 5.
			The node 4 is connected to the port 3 of the node 5.
The parent-set is empty, so the tree is built.			

As this example shows, with just the information contained in the self_id packets one can build the connection tree.

3.2.4.1.3   Translation Table Construction

The aim of this stage of the algorithm is to build the translation table of the node numbers which give the new node numbers after the reset.

Information available initially is:

▧ trees built with the above algorithm before and after the reset but with the current node (which realizes this process) as root,

▧ the old number of the current node and its new number,

▧ normally a node cannot be swapped with another without producing at least 2 resets: so nodes can only appear or disappear in a network, not move.

We assume that a node will be a child of another node if it is connected to it and it is more distant than its parent from the root.

The main principle of the algorithm is that on each port:

※ before a reset, there is the node number A, and after the reset there is the node number B, so the new number of A is B,

※ before a reset, there is the node number A, and after the reset there is no node, the node number A, and all the children of this node, have been removed from the network,

※ before a reset, there is no node, and after the reset there is the node number A, the node number A, and all the children of this node, have been added to the network.

The following steps are performed for all ports of all nodes. It begins with the local node which serves as a common point in the before and after connection trees.

The basic function is *ProcessNode (old, new)* which processes the node numbered *old* before the reset and *new* after.

1. if *old* differs from *NONUMBER*, go to step 6

2. take the first port

3. if there is a child numbered C on this port, add *new* to *added-node-set* and call *ProcessNode(NONUMBER, C)*

4. if there is another port, go to step 3

5. go to step 13

6. put the relation between *old* and *new* in the *translation-table*.

7. take the first port

8. if there is no child on this port in the old and in the new tree, go to step 12

9. if there is no child in the new tree, add the old child number (with all its sons) to *removed-node-set* and go to step 12

10. if there is no child in the old tree, add the new child number (C) to *added-node-set* and call *ProcessNode(NONUMBER, C)*

11. else call *ProcessNode(*the old child number, the new child number*)*

12. if there is a next port, take it and go to step 8

13. end of this node process (all its sons have been processed).

Note: The NONUMBER is used to indicate to the function that the node given in new was not present before the reset and so has no old number, it allows processing of its children which are automatically new too.

An improvement for better security can be added before point 6: to help assure that *old* and *new* are the same node, one can verify that their features are the same (number of ports and speed capability).

3.2.4.2   Message Size Guidelines

Since all TAM packets for the current message will be sent before processing the next pending message for the same destination node (see section 3.2.2) very large messages should be used with caution. A very large message will block important global events and/or messages of other software elements. It may cause a "freezing period" that disrupts the functioning of software elements on the same node – even software elements that have no relation to the sender of the very large message.

For example, even in the best case of a message sent by 1394 asynchronous writes of 512 byte packets on a 100 Mbps network, the full length message (0xffff ffff bytes) will take 343.6 seconds (5.7 min) to transfer. This time is likely too long to wait for the next message, moreover it breaks the timeout rule for the Messaging System (30 sec).

In the worst case, when 63 nodes are simultaneously sending full length messages on the 100 Mbps network with 80% isochronous traffic, the transfer will take 36.7 hours. In this worst case, the largest possible message size which does not break the timeout is approximately 960 Kbytes.

(*Note* – these examples are theoretical and the practical performance may be lower.)

For safety, HAVi makes the following recommendations regarding HAVi message sizes:

※ to not break the Messaging System timeout, it is recommended to use messages of size less than 512 Kbytes.

※ to ensure tolerable response for users, it is recommended to use messages of size less than 64 Kbytes. (This takes about 2 seconds in the worst case.)

Messages larger than these sizes should be used with caution. The bulk transfer APIs (see section 5.14) are recommended if message size exceeds 64 Kbytes.

Messages larger than 64 Kbytes can be refused or aborted by the Messaging System to avoid a timeout. The size which is refused or aborted is Messaging System implementation dependent. It is recommended that the Messaging System makes a best effort to send very large messages.

3.2.4.3   Software Element Design

The basis of a HAVi network is that requests are sent from Software Element to Software Element, actions are taken, and corresponding responses are returned. How the Software Element handles these requests, actions and responses is largely up to the designer; however, it is critical that the designer consider the Software Element's use of HAVi messaging carefully.

Received message requests often trigger various actions. If the actions are implemented in a way which blocks the thread that receives new messages, no new message will be processed until the previous action completes. If an action takes a long time to complete, such a design may result in poor performance. Even worse, if a blocking action includes a request to another software element, a deadlock condition may occur. An example of deadlock is when two Registries simultaneously query each other, but cannot receive the queries, since they cannot complete their previous actions until the queries are answered.

In general there are two solutions to the problem: one is to use one or more additional threads to send synchronous requests. Another is to send requests asynchronously. A small number of multiple threads may be acceptable for simple situations, but multiple threads may be costly. Also, in complex situations, it may be difficult to predict the maximum number of threads needed. By implementing asynchronous requests a Software Element can handle complex situations efficiently.

When making an asynchronous request, a Software Element must store the transaction ID and possible other information about the original request in some sort of table. Normally the entries are removed every time a matching response is received. In the case that a matching response is not received, the corresponding request information will not be automatically removed from the table, causing a potential memory leak. Worse yet, if the SE never receives the response, it may not be able to complete its action, causing parts of the HAVi system to freeze.

To avoid these problems it is recommended that Software Elements that implement asynchronous requests also implement some sort of a timeout mechanism. The timeout mechanism would ensure that actions are completed, and that request data does not build up in tables. (Note that for synchronous requests, such a timeout mechanism is provided by the Messaging System.)

There are many reasons that may cause responses not to be received. One reason may be that the destination Software Element, or destination device, is removed during a request. In these cases a timeout will occur, completing the request. Unfortunately, the timeout may not occur until after a long delay, resulting in very long response times.

In order to respond more quickly to removed devices or software elements, the Software Element designer may choose to activate a watch (msgWatchOn) on the target before sending a request message. Another option would be to detect the disappearance of the target by registering for events such as NetworkReset, GoneDevices, and GoneSoftwareElement.

3.2.4.4　Unknown source GUID / node ID (informative)

A HAVi device may receive a TAM packet/message with an unknown source GUID and unknown source node_id (the device has not been detected within the bus or the IEEE1394 bus_ID indicates that the packet originates from a remote bus).

To allow answering to such a message, the following option may be implemented:

The incoming message will be processed and dispatched by the message system. If the incoming message is a request, then the GUID-node_ID association is temporarily kept by the target device in order to be able to send back a response. Once the HAVi response message is sent, the temporary GUID-node_id association is removed. Note that this temporary GUID/node_id entry is private to the messaging system and not known to the CMM. For this reason it is not reflected in the GUID list and no device changed events are posted.

3.3 Event Manager

The Event Manager provides an event delivery service. An event is a change of state of a software element or of the home network. For example, adding or removing a device implies a change of state of the network and is likely (but not necessarily) to trigger an appropriate event. The delivery of an event is done either locally within a single device or globally to all devices in the network; local or global delivery is selectable by the event poster. To support this service, the Event Manager functions as an agent to help assure the event posted by a software element will reach all software elements that care about the event. If a software element wishes to be notified when a particular event is posted, it must register such intention with its local Event Manager. Each Event Manager maintains an internal table containing the list of events registered by software elements. When a software element posts an event, it does so via a service provided by Event Manager. The Event Manager checks its internal table and notifies those software elements that have registered this event. Software elements that do not register the event will not receive a notification. If the event is posted globally, the local Event Manager also relays the event to all remote Event Managers in the network. Each remote Event Manager performs the same lookup and notifies the registered local software elements. An Event Manager notifies software elements by using the HAVi Messaging System; in particular, it sends a notification message to the software element that is to be notified.

An event has an optional buffer that can be used to pass information related to the event. For example, consider an input device with multiple buttons, a button pressed event could be generated when the user presses a button. The software element that is notified of this event may also be interested to know which button was pressed. The event poster can optionally put additional

information in the buffer and let the Event Manager pass this information to other software elements. A software element would get the optional information as part of the notification process and it is the software element's task to interpret the information.

3.3.1 Mapping IDL Events to the Event Manager API

The specifications of the HAVi software elements contain IDL definitions of events they generate. Events are described by an IDL procedure with only input parameters and a `void` return value. In general they have the following format:

```
void EventName ( in param_type1 param_name1,...) ;
```

When a software element posts an event it has to map the IDL event definition to parameters of the `EventManager::PostEvent` API. The rules for mapping IDL event definition are as follows:

▨ The IDL procedure name `EventName` should be mapped to the `EventId` *event* parameter of the `PostEvent` API.

▨ The `global` parameter must indicate the distribution of the event as given in Annex 11.9.

▨ All event parameters must be provided in the `eventInfo` parameter. They should be encoded using the Common Data Representation (CDR) standard in the same order they are given in the event definition. The first byte of `eventInfo` is considered the "zero index" for natural boundary alignment.

3.4 Registry

The Registry is a system service whose purpose is to manage a directory of software elements available within the home network. It provides an API to register and search for software elements. The Registry service shall be present on each IAV and FAV. Within one device any local software elements can describe itself through the Registry. If a software element wants to be contacted, it must register with the Registry. System software elements shall be registered so that they can be found and contacted by any software element in the network.

The Registry maintains, for each registered object, its identifier (SEID) and its attributes. The Registry also provides a query interface which software elements can use to search for a target software element according to a set of criteria.

3.4.1 Registry Database

Each Registry contains tables describing local software elements (software elements within the same device). The logical database is viewed as the set of all these tables. Each Registry service offers the possibility to query this database.

Each Registry database has the structure indicated in Table 5:

Table 5.   Registry Database Structure

Syntax	Number of bits	Identifier

Database() {
for (e=0; e<N;e++){
      SEID
for (a=0; a< M; a++) {
	80	uimsbf

`        attribute()`
`    }`
`  }`
`}`		

`    attribute()`

Element	Description
SEID	The *software element identifier* is an 80 bit number representing the unique identifier of a software element within the home network.
	It is provided to the software element by the Messaging System API. Any software element can send a message to another using this software element identifier.
Attribute	The *software element attributes* that characterize the software element. All attributes can be gathered in a table and have the structure indicated in Table 6.
N	The max number of entries (registered software elements) in the database.
M	The max number of attributes for a software element

3.4.2 Registry Attributes

A Registry attribute has the following structure:

Table 6.   Registry Attribute Structure

Syntax	Number of bits	Identifier

Attribute() {
    Class
    Name
    Size
for (j=0; j<Size;j++){
        value[j]
    }
}
	1
	31
	32
	
	8
		uimsbf
		uimsbf
		
		uimsbf

Element	Description
Class	The *attribute class* is private (1) or system (0). In case of a system attribute the name values (see below) are well defined. In case of private attributes, the name must be associated with some other attribute like Device Manufacturer or Software Element Manufacturer. This means that a query on a private attribute has to contain some other criteria to avoid private name conflicts.
Name	The *attribute name* indicates the name of an attribute. It is represented by a number. The next section presents the system attribute list. An IDL type is associated with each *system* attribute (see Table 7 below), this type indicates allowable values for the attribute.
Size	The *attribute size* gives the number of bytes of an attribute value.

valuej	valuej indicates the jth byte of the attribute value. The attribute value is formatted as specified in the CDR transfer syntax ([5] – chapter 12.3) with the following restrictions:
	(1) The byte ordering is BIG ENDIAN: MSB (Most Significant Byte) first.
	(2) Since the types of the arguments are assumed to be known by the caller, the attributes can be unambiguously interpreted and CDR type codes are not used.
	(3) The fixed size for the wchar IDL type is 2 bytes according to UNICODE UTF-16 and ISO 10646 UCS-2.
	(4) The first byte of the attribute value (value0) is considered the "zero index" for natural boundary alignment.

The following table identifies the predefined system attributes.

Table 7.   Predefined Registry Attributes

Attribute Name	IDL Type	Fixed or Variable Size	Max size (bytes)	Presence	SV or MV
ATT_SE_TYPE	SoftwareElementType	F	4	M	SV
ATT_VENDOR_ID	VendorId	F	3	M[A]	SV
ATT_HUID	HUID	F	28	M[A]	SV
ATT_TARGET_ID	TargetId	F	18	M[A]	SV
ATT_INTERFACE_ID	InterfaceId	F	2	M[A]	SV
ATT_DEVICE_CLASS	DeviceClass	F	4	M*	SV
ATT_GUI_REQ	GuiReq	F	4	O	SV
ATT_MEDIA_FORMAT_ID	MediaFormatId	F	8	O	MV
ATT_DEVICE_MANUF	DeviceManufacturer [3]	V	106	M*	SV
ATT_DEVICE_MODEL	DeviceModel	V	106	O	SV
ATT_SE_MANUF	SoftwareElementManufacturer	V	106	O	SV
ATT_SE_VERS	SoftwareElementVersion	F	4	M[S]	SV
ATT_AV_LANG	AvLanguage	F	4	O	SV
ATT_USER_PREF_NAME	UserPreferredName	V	38	M*	SV

symbol	meaning	symbol	meaning
M	Mandatory	F	Fixed size
M*	Mandatory for DCM and FCM	V	Variable size (up to the maximum size)
M[A]	M* + Application Module	M[S]	M* + system components
O	Optional	SV/MV	Single valued/Multi-valued

The set of attributes associated with a software element in the Registry database may contain at most one occurrence of those attributes that are "single valued" (indicated by SV in the above table).

---

[3] The size of a wide char (wchar in IDL) will be two bytes (see "Messaging System "chapter).

3.5 Device Control

In a HAVi network, a DCM (Device Control Module) should exist for each device known in that network. The DCM provides an interface to the device by presenting it as a software element in the HAVi architecture. Associated with a DCM are zero or more FCMs (Functional Component Modules). FCMs are software elements that represent the different functional components contained within a device. The number of FCMs within a DCM is flexible and may vary over time. A DCM can be asked for the list of FCMs it currently contains.

A DCM may also use an FCM to represent the functionality of an *external legacy device*. (Such a device is one connected via an external plug to the device represented by the DCM itself.) How the DCM identifies the external device is proprietary to the DCM. It is recommended that DCMs do not accept connection requests on external plugs that are connected to external legacy devices "replaced" by FCMs.

Applications can query the Registry to find the devices and functional components available, and to obtain their software element identifiers. This allows the application to interact with the device via the DCM and the FCMs. A DCM and its FCMs are obtained from a DCM code unit for the device. DCM code units are installed by FAVs and IAVs. Installation of a code unit results in the installation of the DCM and all the associated FCMs. DCM code units can be written in Java bytecode, in which case they can be installed on any FAV device, or in some native code, in which case they can be installed only on (and by) some FAV or IAV that can execute that code. More concretely:

- *DCM code unit.* A piece of code related to a HAVi device. DCM code units are handled and installed by FAV and IAV devices. When a DCM code unit is installed, the DCM code unit will in turn install the DCM for the device; the DCM in its turn will install the FCMs for the functional components currently available within the device. DCM code units can be written in Java bytecode or a native code. DCM code units may come from different sources (e.g., embedded in an IAV, stored in a BAV, or from the Internet). The format of DCM code units is described in section 7.4.1.

- *Device Control Module (DCM).* A software abstraction of a device providing device specific functionality to the HAVi software environment and applications. HAVi applications will not communicate with a device directly but through the DCM of the device (or one of the FCMs). A DCM is an HAVi object in the sense that it is registered in the Registry and it can communicate with other HAVi objects via the HAVi Messaging System.

- *Functional Component Module (FCM).* A software abstraction of a functional component providing the functionality of that functional component to the HAVi software environment and applications. HAVi applications will not communicate with a functional component directly but only through the FCM (this is at least the model used to present the relation, the FCM implementation may communicate with the CMM directly). An FCM is an HAVi object in the sense that it is registered in the Registry and it can communicate with other HAVi objects via the HAVi Messaging System.

For the different types of HAVi devices, DCMs play a different role.

- An IAV device may host one DCM representing itself and may host one or more DCMs representing LAVs (or BAVs operating in *LAV mode*, see below).

- An FAV device shall host one DCM representing itself and may host one or more DCMs representing LAV devices and/or BAV devices.

▓ An LAV device does not have any notion of DCMs. When attached to a HAVi network where one or more FAV or IAV devices know how to handle the LAV, one of them has to provide the DCM code unit to make the LAV available to other HAVi components. How this is done and how the DCM/FCMs communicate with the LAV device is completely proprietary to the manufacturer of the FAV or IAV device.

▓ A BAV device does not host any DCMs, but provides a DCM code unit in Java bytecode. When attached to a HAVi network with one or more FAVs, one of them uploads and installs the DCM code unit to make the BAV device available to other HAVi components. Installation of the DCM code unit results in the installation of the DCM and all FCMs related to the device. In this situation, the BAV is said to operate in BAV mode. The IEEE 1394 communication between the DCM/FCMs and the BAV goes via the standardized CMM1394 API. However the content of the IEEE 1394 messages interchanged by the DCM/FCMs and the BAV is proprietary to the BAV manufacturer.

▓ When attached to a HAVi network with no FAV devices but with an IAV device that knows how to handle that BAV, an IAV device can provide a DCM code unit itself to make the BAV device available to other HAVi components. The BAV is then said to operate in *LAV mode* in which case the situation is the same as for an LAV device.

Besides the APIs to control the device (and its functional components), a DCM may also contain a device specific application. Through this application, a device manufacturer can provide the user a way to control any special feature of the device in a way decided by manufacturer, without the need for standardizing all these features in HAVi. It is provided via the API of the DCM and may be provided at two different levels. For Level 1 interoperability, the DCM may provide an API for the Data Driven Interaction mechanism. For Level 2 interoperability, the DCM provides an API for FAV devices to upload a havlet code unit. Such a code unit consists of bytecode that can be installed by an FAV device and results in a havlet. The format of havlet code units is described in section 7.4.3.

A havlet obtained from a DCM is a Level 2 (i.e., Java bytecode) application that provides a user interface for control of the device associated with the DCM. For the actual control of the device, the havlet communicates with the DCM (using its standard interface, and possibly, proprietary extensions). A havlet is a HAVi object in the sense that it can communicate with other HAVi objects via the HAVi Messaging System.

A havlet code unit is always uploaded and installed on the initiative of an FAV device. An example scenario is shown in Figure 18.
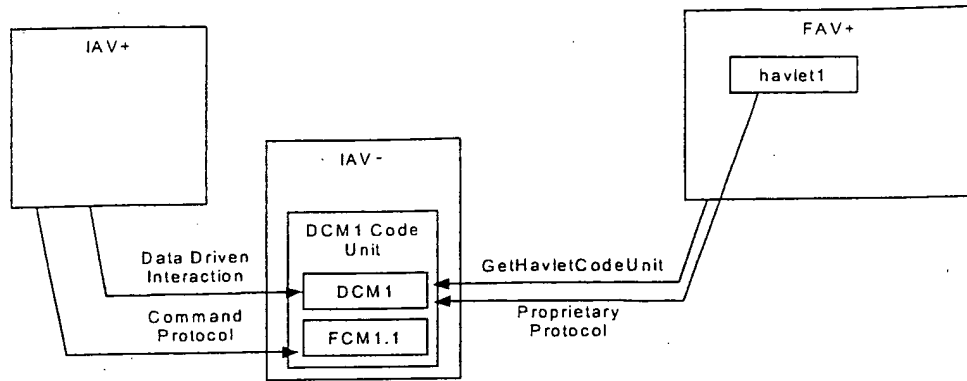
Figure 18. Havlet Upload

The figure above shows an IAV- (without a display) that provides a DCM for some device. An IAV+, with a display, can control that device in the following way. An embedded application on IAV+ interacts with the Functional Component Module *FCM1.1* (or Device Control Module *DCM1*) through HAVi messaging. For device specific functionality, IAV+ can allow user interaction with *DCM1* through the Data Driven Interaction mechanism. An FAV+ could handle the IAV- in the same way. But as shown in the picture, an FAV+ also has the ability to upload a havlet. Therefore it interacts with *DCM1* to get the havlet code unit. After installation of this bytecode, *havlet1* is available. To control the device, the havlet communicates with *DCM1* via HAVi messages, however the content of these messages may be proprietary to the manufacturer of the DCM and havlet.

DCM Managers are responsible for installing DCM code units for new devices attached to the HAVi network. A device may consist of more than one functional component; e.g., one device may consist of a tuner and a VCR. For a BAV or LAV device, the installation of its DCM code unit always takes place on a per device basis, not for each DCM component separately, so, for a DCM manager a DCM code unit is a single entity for installation, removal and replacement of DCM components. Furthermore, one DCM code unit corresponds to a single BAV or LAV device, i.e. it contains all DCM components for that device.

For BAV devices, DCM code units play a special role. A BAV device provides a single DCM code unit as a piece of bytecode (as part of the SDD) and a standardized mechanism for communication with an FAV. An FAV device can upload and install a DCM code unit (by the DCM Manager). Installation of a DCM code unit results in the instantiation of all DCM components representing that BAV device.

The communication relation between the components of a BAV DCM code unit and the system components of its hosting controller are depicted in the figure below. The CMM provides a basic service for elementary communication with the BAV device. The DCM Manager detects the attachment of a new device and obtains the GUID of that device; it can then communicate with the device to obtain basic information about the device and to retrieve the DCM code unit. The DCM Manager installs (and later removes) the DCM code unit. This installation results in the DCM and FCM objects that communicate with the device via the CMM. On installation, DCMs and FCMs make themselves known via the Registry so that they can be used by other applications via the HAVi Messaging System.
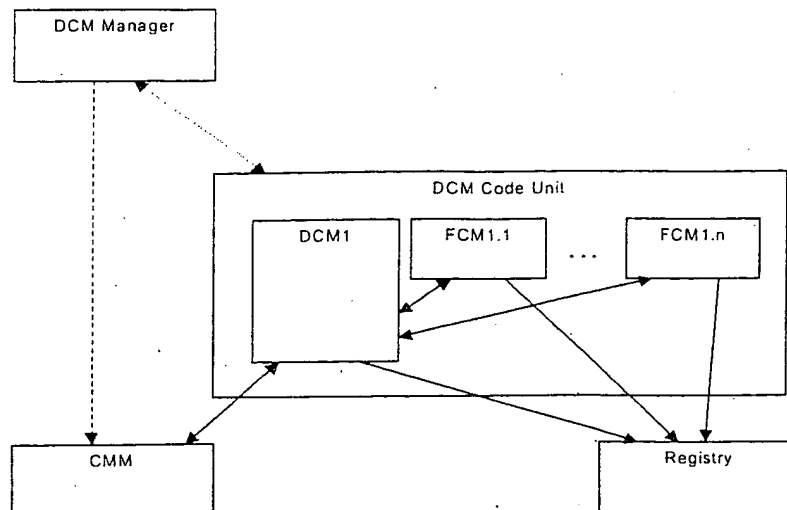
Figure 19. DCM Installation

The communication between the DCM components and the CMM is based on the GUID of the BAV/LAV device. The CMM requires only the GUID for communication with the device driven by DCM components. The CMM does not need to have any knowledge about the structure of functional components within a BAV device. This simplifies the communication between an FAV and a BAV device, makes the standardization effort smaller, and increases the possibilities for using proprietary protocols. The DCM components themselves are responsible for the proper use of the CMM, i.e. for interleaving the communication in a proper way and the distribution of messages received from the BAV device to the proper DCM components.

The contents of BAV DCM code units must be standardized so that each DCM Manager can handle DCM code units from arbitrary BAV devices. The DCM code unit is a kind of self-extracting package in Java bytecode; it provides the DCM Manager with handles for installation and removal. The DCM Manager just calls the install handle and provides the device's GUID (for future communication between the device and the DCM via the CMM). The DCM code unit itself is responsible for installing all its DCM components. Similarly, the remove handle provides the DCM Manager with a handle to remove all DCM components within the DCM code unit. This allows maximum freedom for BAV manufacturers in structuring their DCM code units.

3.5.1 Device Control Modules

A DCM, is a software abstraction of a device providing device specific functionality to the HAVi software environment and applications. HAVi applications will not communicate with a device directly but through the DCM of the device or one of its FCMs. A DCM is a HAVi object in the sense that it is registered in the Registry and can communicate with other HAVi objects via the HAVi Messaging System. DCMs and FCMs are registered with their type and their HAVi unique identifier (HUID). The HUID allows applications to find the DCM or FCM after partial system unavailability (as when the device represented by the DCM is momentarily removed from the network).

3.5.1.1   General

A DCM provides a set of basic methods for device control and observation. This API can be used by HAVi system components as well as any application. It includes the following functionalities:

▓ Device representation – Provides a (Level 1) visual representation of the device that can be displayed to the user.

▓ HUID information – Provides the HUID of the DCM as well as those of the corresponding FCMs.

▓ User Preferred Name – Allows assigning and retrieving of a user specified device name.

▓ Power management – Provides means to turn the power of the device on or off.

▓ Native Commands – Provides means to control the device in its native command set (CAL, AV/C, etc).

3.5.1.2 HAVi Unique Identification

In HAVi, it is possible to write applications and system components (like a Resource Manager) that always use the same functional component or that look for the same functional components used in a previous session. To support this capability, applications must have a unique and persistent way to identify the DCM representing a device and to identify the FCMs representing functional components on that device. This identification is called the HAVi Unique ID, or HUID. Ideally the HUID would be persistent across bus resets and network reconfigurations, however this is not guaranteed under all circumstances and a DCM for a certain device may be assigned different identifiers at different times.

For example, it is possible to write an application that sets up a user preferred configuration for watching a video: using the small TV set, the upper VCR deck of the 3 deck VCR box and the special wireless headphone set, all situated in the living room. This configuration would be indicated by the user once, and each time the application runs, the specified configuration must be set up.

Note that identification of DCMs and FCMs is subtly different from identification of devices and functional components. The identification of a DCM/FCM indicates the device as well as the functionality of that device provided by the DCM/FCM. This difference can be seen by the following example:

Assume a network with a BAV CD-ROM player, an FAV and an IAV device. The DCM code unit of the BAV device is uploaded and installed on the FAV device and provides the full functionality of a CD-ROM player. The DCM code unit gets a HUID based on its 1394 GUID. An application running on the IAV device stores this HUID and when it wants to use the CD-ROM player it can retrieve the DCM based on this HUID (when it is available in the network). Now, the FAV device is removed. The IAV is able to control the CD-ROM player, but has only a DCM that provides the CD-Player functionality of the CD-ROM. This DCM should not get the same HUID as the BAV's DCM since the functionality provided by the DCM is different (although it represents the same device). This prevents an application from mistaking this DCM for the DCM of the CD-ROM player. However, instead of getting a CD-ROM, the application now only gets a CD-Player. (In this situation the difference can be easily detected by the type of the DCM, however, the problem may be more subtle in general.)

For FAVs, IAVs and BAVs (in BAV mode) and their functional components, the HAVi Unique Identifiers allows a unique and persistent identification, which means:

Unique – By the HUID, an application is able to detect:

▓ Whether a device or functional component is the same as a device or functional component it has used before: if a software element with HUID *H* represents the device or functional component *D*, then it will *always* be the case that any software element with HUID *H* represents *D*.

This means that the HUID must be able to identify the device (the black triple deck VCR in the family room, instead of the VCR in the kitchen); as well as the functional component on that device (the upper VCR deck instead of the middle or lower one).

▓ Whether a DCM or FCM is the same as a DCM or FCM it has used before: if a software element with HUID *H* has some API *A*, then it will *always* be the case that any software element with HUID *H* has API *A*.

This means that it also should be able to identify the specific API corresponding to the device or functional component. In general, a device can be represented by different DCMs and FCMs (at different points in time) which may have different APIs; e.g., different proprietary extensions of the standard command sets.

Note that the type (of device or functional component), Vendor ID (of the device) or GUID or even a combination of these is not sufficient to uniquely identify an FCM. There may be several identical (of the same type and vendor) devices in the network containing several identical functional components (e.g., a double deck cassette player, or a triple deck VCR). .

Persistent – The HUID is persistently unique over network resets, device removal, re-attachment as well as complete network shutdowns: if a device or functional component *D* is represented by a software element with HUID *H*, then it will *always* be the case that *D* is represented by a software element with HUID *H*.

Consequently, with the HUID, an application is able to look for the FCM of a specific functional component even when the network configuration has changed, and the functional component may be represented by another FCM on another FAV/IAV device with a new HAVi SEID.

Unique and persistent identification of FAVs, IAVs and BAVs (in BAV mode) is possible due to the following facts:

▓ Each FAV, IAV and BAV device supports IEEE 1394 and IEEE 1212, so the device can be identified uniquely and persistently by the GUID.

▓ The DCM code unit and the FAV, IAV, or BAV (in BAV mode) device are always from the same manufacturer (or at least manufacturer compliant) as the device itself. So, the device manufacturer can (and has to) take care that each functional component always has the same tag.

However, for LAV devices (or BAV devices in LAV mode) it may not always be possible to identify a functional component on an LAV device persistently for the following reasons:

▓ The device cannot be uniquely and persistently identified. An LAV does not need to support 1212 (defining the IEEE 1394/1212 GUID) or may even be connected via some other physical bus (e.g. USB or SCART). So, the GUID cannot be used for their unique identification. Although it may be the case that there are other ways to identify the device, it may also be the case that such LAV devices cannot be uniquely identified at all (since two devices can be completely identical and their location or connection point to a bus cannot be determined).

▓ The device can be uniquely defined, but the DCM representing the device may differ (after network re-configuration). Then, it might not be possible to determine the proper FCM of a specific functional component within a DCM. This identification is the responsibility of the DCM code unit, which is, for an LAV, provided by the IAV/FAV that hosts the LAV. Since the communication between IAV/FAV devices and the LAV device is not (HAVi) standardized, different IAV/FAV manufacturers may select different identifications of FCMs. E.g., for an AV/C-CTS device with a tuner and a VCR, an IAV of Vendor A might provide a DCM in which the FCM of the tuner is identified by tag 1 and the FCM of the VCR by 2, while an IAV of Vendor B has a DCM in which the VCR is identified by 1 and the tuner by 2.

Although it might not always be possible to uniquely and persistently identify FCMs of functional components of LAV devices in a general way, there are many situations in which it can be done. For example:

▓ Assume a network with an IAV and one LAV that can be identified uniquely and persistently. Since the IAV can identify the LAV each time it installs its DCM code unit, it can assure that the same DCM code unit is always installed. So, this specific DCM could be identified by the GUID of the IAV, combined with an identification of the LAV.

▓ Assume a network with two IAVs and one LAV that can be uniquely identified, all from the same manufacturer. The manufacturer can assure that the DCMs for that LAV are the same on both IAVs and that both DCMs (and FCMs) can be identified by a vendor specific identification.

▓ Groups of manufacturers making devices for some standard, e.g. AV/C-CTS or CAL, may agree on how these types of devices should be embedded in HAVi. When DCM manufacturers follow this embedding, it can be guaranteed that the DCMs (and FCMs) for these types of devices are similar and obtain the same identification.

So in general, HUIDs for LAVs (and BAVs in LAV mode) are not always unique and persistent. However, HUIDs will exploit as much as possible the situations described above. Each DCM and FCM stores its HUID in the Registry. DCMs and FCMs can then be retrieved uniquely from the Registry. An application can choose to store locally any information, such as attribute values, it retrieves from the Registry. When the application wants to find a DCM/FCM again, it can query the Registry with these attributes and retrieve the SEID of the DCM/FCM.

3.5.1.3   User Preferred Name

To allow a more user friendly reference to a device, it is possible to assign a "user preferred name" to a DCM. This might be something like "John's VCR", "The Red Box", or "The VCR downstairs". It is a system wide name that the user can rely on for identifying a thing (e.g. the one with the smart card slot) with that name. The name of a DCM can be retrieved via the DCM API, and can also be found in the Registry. The name can be modified via a method of the DCM API. Quite likely a graphical user interface for the DCM would provide a user friendly way to fill in this name. On modification of the name, the DCM is responsible for modification of the Registry entry to keep the naming consistent. For user convenience, it would be best if the DCM could store this name persistently (e.g. in some non-volatile memory on the device), so that the same name can be used over time, even if the device is switched off and on. However, this may not be the case. The user preferred name may be lost in case of a power off, or a DCM reinstallation (after a network reset) and the user would then need to assign the user preferred name again. Consequently, a DCM is allowed to provide the empty string as the user preferred name.

The user preferred name is meant as an aid to the user for identification. Therefore, the consistency of the usage is also in hands of the user, and HAVi does not provide any means to check or guarantee these consistencies.

3.5.1.4 Native Commands

HAVi has been designed to support the embedding of non-HAVi devices in the HAVi framework. To allow extensive control of these devices, HAVi provides a way to pass native commands to the device. This could be a command native to that specific device, native to the vendor, or native to other non-HAVi standards (e.g. CAL, AV/C). A native command may have side-effects on the standard HAVi DCM service or the service of one of its FCMs. It is the responsibility of the DCM (and its FCMs) to assure that the HAVi standard interface is not violated and to determine whether this specific native command is accepted or not. E.g., a DCM may receive a AV/C "play" request for a VCR sub-unit that is controlled by one of the FCMs. It is the decision of the DCM as to whether the native command is executed when the FCM has been exclusively reserved by another application.

3.5.1.5 Connection Management

DCMs also provide a high level API to allow other objects to query the state of connections within the device and to manipulate those connections. This API is used by the Stream Manager. The connection management part of the API allows device connections to be established, both internally between functional components, and from functional components to the external network. Connection status can be queried and connection capabilities (transmission formats) can also be queried.

A user may directly manipulate a device through a Level 1 or Level 2 interaction. If such an interaction starts, it is possible that connections within the device already exist. In that case, the interaction should normally keep these connections in place. Any DCM or device activity generated by the user interaction may result in certain streaming behavior through these connections. If no such connections exist, but are required at some time, the DCM can establish them, possibly through interaction with the user.

3.5.1.6 Level 1 User Interaction

A DCM may provide a Level 1 device specific user interface via a dedicated set of API primitives as described in section 5.12 on APIs for Data Driven Interaction. The DCM indicates in the Registry whether this form of user interaction is provided.

3.5.1.7 Level 2 User Interaction

A DCM may also support a Level 2 solution by providing a havlet. FAVs can upload, from the DCM, the bytecode for the havlet. A DCM that provides a havlet indicates this capability in the Registry.

3.5.1.8 Resource Management

DCMs can be involved in reservations and scheduled actions. Section 3.8 on Resource Management describes the consequences of this involvement for DCMs and FCMs. A DCM has to take care of "dependent FCMs" in the same DCM code unit that are bound to other FCMs for some reason, and that must always be "internally reserved and released" together with those FCMs.

If a DCM also supports the DDI protocol, it is the responsibility of the DCM acting as a DDI Target to ensure that a DdiTarget::UserAction message does not result in reservation violations. This can be accomplished, for instance, by the DDI Target not "showing" facilities (DDI elements) that are reserved by other applications. It also indicates that a reservation is not considered as "part of" the DdiTarget::Subscribe. It is up to the application whether or not it reserves one or more (or all) FCMs before the subscription. It should be noticed that "unreserved" FCMs are free for use by any application and thus also via a DDI Controller. If reservations are done by the subscribing application, the DDI Target of course should take them into account for allowing UserActions of that application.

3.5.2 Functional Component Modules

An FCM is a software abstraction of a functional component providing the functionality of that functional component to the HAVi software environment and applications. HAVi applications will not communicate with a functional component directly but only through the FCM. An FCM is a HAVi object in the sense that it is registered in the Registry and it can communicate with other HAVi objects via the HAVi Messaging System.

HAVi has defined the following FCMs: tuner, VCR, clock, camera, AV disc, amplifier, display, AV display, modem, and Web proxy.

3.5.2.1 General

An FCM provides a set of basic methods for device control and observation. This API can be used by HAVi system components and applications. It includes the following functionality:

- HUID information – Provides the HUID of the FCM as well as the corresponding DCM.

- Type information – Within HAVi a set of functional component types has been identified, e.g. VCR, tuner, display, etc. For each HAVi defined type there is a HAVi defined message set specifying an API for the control and observation of such a device. This API gives the type of the functional component represented by this FCM and indicates which HAVi messages are supported by this FCM.

- Power management – Provides means to change the power state of the functional component from "on" to "stand-by".

- Native Commands – Provides means to control the device in its native command set (CAL, AV/C, etc) and is similar to the interface on the DCM.

3.5.2.2 Notifications

For certain events, software elements may like to subscribe directly to the event source since it knows from which software element the event will be generated and it is only interested in an event from that source. In the case of a VCR for example, when an application subscribes to an "end-of-tape" event, it knows exactly from which VCR the event is expected. It is not interested in receiving end-of-tape" events from a VCR it is not controlling.

Moreover, for some types of events, the event subscription must be flexible enough to allow conditional event programming: in a large number of cases, it may be interesting to subscribe to a conditional event (i.e., an event is generated if the condition is matched). Generally, functional components have a certain number of state variables that they make available via their API. It

would be restrictive to define *a priori* which of these variables shall be able to generate events, and which not. Moreover, it would be nearly impossible to define which state transitions may be interesting for future applications, and which not. So, proposing a special event for each imaginable combination would result in a huge and unwieldy API specification.

For this reason, a general mechanism, called "notifications", is provided. Two methods of the FCM API allow an application to subscribe and unsubscribe to notification of specific changes of a specific attribute. By a relational expression ("attribute is bigger than 10"), the application can indicate what kind of change it is interested in.

The general FCM API only provides the general part of this API, i.e. the methods for subscribing and unsubscribing. It does not provide the specific sets of attributes. Those should be provided by the specific functional component APIs.

Although the mechanism of notifications is described as a part of an API for an FCM, this mechanism could also be a part of the API of any software element capable of generating events.

3.5.2.3   Connection Management

FCMs also provide a high level API to allow other objects to query the state of connection plugs on the FCM that may be involved in internal or external connections. This API is used by the Stream Manager. The connection management part of the API allows queries about connection status and connection capabilities (stream types).

3.5.2.4   Resource Management

FCMs can be involved in reservations and scheduled actions. Section 3.8 on Resource Management describes the consequences of this involvement for DCMs and FCMs. An FCM is not obliged to support reservations if it only has non-state changing methods. In this case, a "not supported" return value will be returned if an application tries to reserve it (and possibly for other reservation-related methods). However, each FCM with state-changing methods shall support reservation facilities.

3.5.2.5   Virtual FCMs

HAVi FCMs typically are interfaces to functionality provided directly by hardware. However it is also possible for an FCM to encapsulate software functionality. This is similar to the idea of a *virtual device*, a device realized, at least partially, by software mechanisms. Based on this analogy, the following two categories of FCM are defined:

- *physical FCM* – an FCM which controls the operation of a functional component of a target device. When a HAVi message is sent to a physical FCM it results in implementation dependent communication between the FCM (or its DCM) and the target device.

- *virtual FCM* – an FCM which controls the operation of software-based processes. When a HAVi message is sent to a virtual FCM, the message is processed internally by the FCM and does not necessarily involve communication with other devices on the network.

FCMs in general present both a control interface (the set of HAVi messages to which they respond) and a content interface (a set of plugs). In order to allow virtual FCMs the same range of functionality as physical FCMs, they must be capable of presenting both control and content interfaces.

62

3.5.3 Havlets

A havlet is a Level 2, device-dependent user application. A havlet is typically a proprietary application that offers a user interface for the control of a specific target device. For the actual control of the device, the havlet makes use of the DCM of the target device. A havlet is a HAVi object in the sense that it is registered in the Registry and can communicate with other HAVi objects via the HAVi Messaging System. Since a havlet is a Level 2 concept, a havlet only runs on an FAV device (because that FAV device has uploaded and installed the havlet code unit). A havlet offers an interface to the user on the FAV device where it resides.

3.6 Device Control Module Manager

The DCM management system is responsible for installing and uninstalling DCM code units for the control of BAV and LAV devices that are directly connected to a HAVi 1394 network. DCM management is collectively performed by the DCM Managers on FAV and IAV devices. DCM code units for FAV and IAV devices are managed by the devices themselves in a proprietary manner. Non-1394 devices shall be managed by FAV or IAV devices in a proprietary manner.

Each FAV device has a DCM Manager, but for IAV devices a DCM Manager is optional. An IAV device is not required to participate in the DCM management process if it will not host any BAV or LAV device on the HAVi 1394 network. (It shall be indicated in an IAV's SDD whether it has a DCM Manager or not – see section 9.10.4.2.)

In the sequel, a BAV or LAV device will be termed a *guest*. Each guest shall be facilitated by a DCM code unit installed on an FAV or IAV device, termed a *host*.

A DCM code unit may be a Java code unit to be loaded and installed on an FAV device, or a native code unit to be installed on an FAV or IAV device. Installation of a DCM code unit results in the installation of DCM and FCMs which shall register themselves.

Each DCM Java code unit is accompanied by a *profile*, which includes the values of the transfer and installation size parameters needed by the code unit and its components (see section 9.10.7). The data in the profile are needed by a host to determine whether it can upload and install the corresponding DCM Java code unit.

Each DCM Manager offers a number of methods that can be invoked by software elements, and a number of global events that notify the results of DCM code unit installation and uninstallation. Most DCM management activities are triggered by a network reset event, which is typically generated when the network topology changes or a device is (de)activated. The network topology changes if, e.g., a device is added to or removed from the network, if the network is split, or if two networks are joined.

3.6.1 DCM Code Unit Installation and Uninstallation

▨ The actions of the DCM management system as the result of a network reset event are discussed next. These actions can be fine tuned or overruled by *preferences*, which can be set locally on certain DCM Managers (discussed in section 3.6.2).

▨ For each guest there shall be at most one installed DCM code unit in the entire network. Due to resource limitations, it may occur that no DCM code unit is installed for some guest on the network. It may also occur that a DCM code unit uninstalls itself for some reason. Normally, another host will be selected for a guest's DCM code unit if the previous host is removed from the network.

※ A DCM code unit shall be uninstalled if the corresponding guest is no longer available on the network.

If a DCM code unit is to be installed, it shall be according to the following scheme (assuming no preferences are set):

※ If an uploadable DCM Java code unit for the guest is available, an FAV host shall be selected to load and install the code unit. If this fails, any host that can install an embedded DCM code unit for the guest in a proprietary way shall be selected.

※ A BAV device may internally store an uploadable DCM Java code unit and/or a URL for such a unit in its SDD data. If a URL for an uploadable DCM Java code unit is specified in a BAV device, it shall be loaded from the specified location and installed on an FAV host, instead of the DCM code unit contained in the BAV device. If this fails, the DCM code unit contained in the BAV device shall be installed instead. (It is recommended for BAV devices to contain a DCM code unit. If there is no code unit, a URL for a code unit should be contained).

※ All other conditions being equal, a host with the lowest number of installed DCM code units is selected.

※ If a host rejects the DCM, either before or after installation is attempted, the next most suitable candidate is attempted until all possible candidates have been tried.

A URL is specified as `<scheme>://<host>/<path>`. The scheme can be, for example, `http` or `ftp` for IP-based protocols, or `file` for storage media. For URLs in the configuration ROM of BAV devices the `file` scheme shall not be used. The URL shall denote an uploadable DCM Java code unit and/or its profile on the Internet or on a (persistent) storage medium. The validity of URL syntaxes is outside the scope of this document. Note that DCM Managers need not have knowledge of URL syntax and semantics.

A DCM Java code unit and its profile are two separate files, each with their own file name. The following convention applies:

※ DCM Java code units shall have a file name with extension hdc (HAVi DCM Code).

※ DCM Java code unit profiles shall have a file name with extension hdp (HAVi DCM Profile). The format of a ".hdp" file conforms to section 9.10.7.

DCM Java code unit URLs shall be specified without the extensions ".hdc" or ".hdp". The proper extension will only be added when either a profile or a code unit is actually retrieved (see also section 9.10.8).

The DCM management protocol will exploit the URL access capability of devices. A host device may decide to cache a loaded DCM code unit instead of retrieving it from the URL-designated location. DCM Managers shall be able to use URL access capable FCMs in the network, but they can also offer URL access capability in a proprietary manner.

Uploadable DCM Java code units shall be encoded in the format described in section 7.4.1. This applies both to code units contained in BAV devices and those designated by a URL.

3.6.2 Preferences

The activities of the DCM management system are guided by preferences, which can be set by applications. Preferences typically specify deviations from the default installation actions described in the previous section. Methods are available for setting and retrieving preferences on DCM Managers in the system. Preferences are optional; DCM management will function without any set preference. If a preference is set on some DCM Manager, it shall be stored persistently if possible, so that it does not have to be entered each time a device is powered up. The value of a preference that is not set on a host is *unspecified*. Preferences shall be modifiable by the user.

The first three preferences in the following table are associated with either a single guest (by its Global Unique Identifier, or GUID) or with a guest model (by its Vendor Model Identifier, or VMID – see section 5.8.2). For an LAV guest, only the GUID variant is possible, since a VMID is not available for LAV devices. A GUID-based preference for a guest overrules a VMID-based preference for the guest model, without causing a conflict. The fourth preference is not associated with a GUID or VMID.

Table 8.   DCM Installation Preferences

Preference	Type
DCM_PREFER_VENDOR_HOST	boolean
DCM_PREFERRED_HOST	GUID
DCM_PREFERRED_URL	string
DM_PREFERRED_URL_DEVICE	GUID

※ DCM_PREFER_VENDOR_HOST – Designates whether a host of the same vendor as the guest is preferred for installing a DCM code unit. If for any guest or guest model the value is set to True by any DCM Manager, the DCM management system shall give preference to a host of the same vendor. The value used for DCM code unit installation is determined by taking the disjunction (logical OR) of all values from the DCM Managers. If unspecified by a DCM Manager, the value False is assumed. However, a vendor may return a value of True for any of its own guests if the value is unspecified.

※ DCM_PREFERRED_HOST – Designates a specific host that is preferred for installing a DCM code unit for some guest or guest model. The reason for setting this may be performance or reliability considerations.

※ DCM_PREFERRED_URL – A URL designating the location of an uploadable DCM Java code unit and its profile for a guest or guest model. For a BAV guest, this preference is used to specify a DCM code unit to be installed instead of a DCM code unit designated by a URL in the BAV device or the DCM code unit contained in the BAV device. For an LAV guest, it is used to specify an uploadable DCM Java code unit that can be installed on an FAV host. (Note that an uploadable DCM Java code unit cannot be installed for LAV guests for which this preference is not set.)

▓ DM_PREFERRED_URL_DEVICE – Designates a specific host or guest that is preferred for URL access to retrieve DCM Java code units. For example, the user may designate a device capable of Internet access or a device that stores DCM code units. If this preference specifies a host, it should be used to retrieve URL-designated code units. If this preference specifies a guest, it indicates that a DCM code unit should be installed for this guest before installing DCM code units for any other guest. The guest should subsequently be used to retrieve URL-designated code units. Note that the URL access capability of a device that is not specified by DM_PREFERRED_URL_DEVICE may be ignored.

For any preference, it may occur that conflicting values have been set at different DCM Managers. One of the values will be selected arbitrarily. However, for DM_PREFERRED_URL_DEVICE an FAV device takes preference over an IAV device, and a host takes preference over a guest. Only devices that exist in the network are taken into account for this preference. DCM-related preference conflicts are reported in DcmInstallIndication events.

Preferences relating to some guest only take effect when a DCM code unit is to be installed for that guest, e.g., upon network reset events, and upon invoking (un)installation requests. Setting (or changing) preferences relating to some guest does not result in the DCM code unit installation, or re-installation even when a DCM code unit for that guest has already been installed. There is a priority among the preferences to resolve ambiguities. For installing a DCM code unit the following steps are taken, in the order listed. The DCM manager will execute each step in turn, attempting to install the DCM code unit on all hosts valid under the current step, until the DCM code unit is installed successfully on a host, or all the steps listed have been attempted.

▓ If <preferred host> is specified and is an FAV device, and <preferred URL> is specified, try to install the code unit designated by the preferred URL on the preferred host.

▓ If <preferred host> is specified and is an FAV device, and the guest is a BAV device, try to install the code unit designated by the SDD URL in the guest on the preferred host.

▓ If <preferred host> is specified and is an FAV device, and the guest is a BAV device, try to install the code unit contained in the guest on the preferred host.

▓ If <preferred host> is specified and is an IAV or FAV device, try to install a code unit on the preferred host in a proprietary manner.

▓ If <prefer vendor host> is True, try to install a code unit on a host of the same vendor as the guest in a proprietary manner.

▓ If <preferred URL> is specified, try to install the code unit designated by it on any FAV host.

▓ If the guest is a BAV device, try to install the code unit designated by the SDD URL in the guest on an FAV host.

▓ If the guest is a BAV device, try to install the code unit contained in the guest on an FAV host.

▓ Try to install a proprietary code unit on any host.

In case an undesirable DCM code unit installation was performed it is possible to use the DCM Manager methods for explicit uninstallation and subsequent installation of code units. For example,

the home network configuration at some time during the network start-up may be incomplete, leading to premature code unit installations.

3.6.3 Interaction between DCM Code Unit and DCM Manager

A DCM Manager shall only install and uninstall DCM code units on the local device. Once loaded (if applicable), the DCM Manager controls the DCM code unit through Java DCM code unit methods. Each loaded DCM code unit shall offer these methods to the DCM Manager. However, the vendor of a HAVi host device can arrange for interaction between a DCM Manager and an embedded DCM code unit to take place in a proprietary manner.

The DCM code unit method install(nodeId, listener) is invoked by the DCM Manager to install the DCM code unit; it is the first method a DCM code unit shall accept. The DCM code unit components (DCM and FCMs) shall be installed and registered. The DCM code unit method uninstall() is invoked by the DCM Manager to uninstall a DCM code unit if required; it is the last method a DCM code unit shall accept.

After a DCM code unit has uninstalled itself, it signals this to the local DCM Manager through the Java listener method uninstalled(). This call shall only occur after all DCM code unit components have been uninstalled and unregistered. After this call, a new DCM code unit can be installed for the guest, if required. A DCM code unit shall uninstall itself if the DCM manager requests it to do so. However, it can also do this on its own initiative.

The DCM code unit concepts and methods are discussed in section 7.4.1.

3.7 Stream Manager

3.7.1 Objectives

The Stream Manager provides an easy to use API for configuring end-to-end isochronous ("streaming") connections. Connections may be point-to-point or utilize "broadcast" sources or sinks. The responsibilities of the Stream Manager include:

- configuration of both *internal* connections (within a device) and *external* connections (between devices)

- requesting and releasing transport system resources

- providing global connection information

- support of plug compatibility checking

- support for connection restoration after network resets

3.7.2 Design Decisions

- A Stream Manager runs on each FAV device; implementation on IAV devices is required only if applications on the IAV need Stream Manager services.

- A Stream Manager provides connection creation services only to applications running on the same device as the Stream Manager itself.

▨   The Stream Manager API is based on the IEC 61883 plug model and the HAVi functional component model.

▨   IEC 61883 broadcast and point-to-point connections are supported. (*Note*, due to the restrictions specified in 61883.1 on broadcast connections, Stream Managers may not be able to establish broadcast connections for some devices, even though the devices themselves support broadcast connections.)

▨   ·Connections do not cross transport boundaries (i.e, connections have a single "transport type" – see below).

▨   Connections originate and terminate at functional components (rather than devices).

▨   Plugs should satisfy stream type compatibility, i.e. the Stream Manager leaves to the application the problem of configuring stream format converters (functional components which sink a stream of one type and then source a stream of a second type).

▨   The Stream Manager is responsible for requesting and releasing isochronous resources, this includes: IEC 61883 PCRs (plug control registers), 1394 bandwidth and 1394 channels.

▨   A standard naming scheme is used for stream types, transport types and transmission formats. The stream type and transport type naming schemes are specified by HAVi and, for stream types, the naming scheme may be extended by vendors; transmission formats are transport type dependent and specified outside of HAVi.

▨   Implementations of the Stream Manager are required to make efficient use of 1394 resources by using the IEC 61883 "overlay" mechanism whenever possible.

3.7.3 Definitions

connection – a uni-directional data transfer path created by a Stream Manager. Typically used for streaming content. Connections originate and/or terminate at functional components. A connection is either an *internal connection* or an *external connection*. *Non-HAVi connections* refer to data transfer paths created by non-HAVi applications or devices.

internal connection – a connection where data is transferred within a device.

external connection – a connection where data is transferred across a device boundary.

attachment – The segment of a connection which exists between a DCM plug and the plug of one of its FCMs.

device connection – an internal connection or an attachment.

connection identifier – Stream Managers and applications refer to connections via unique identifiers. These values persist throughout the lifetime of the connection. Connection identifiers for connections created by Stream Managers are globally unique and their reuse should be avoided as much as possible.

stream type – identifies a media representation, this may be a data format (for digital media) or signal format (for analog media), e.g. CD audio, composite video.

transport type – identifies a transport system. HAVi compliant implementations of the Stream Manager must support three transport types: IEC61883 for IEC 61883 connections running over 1394, INTERNAL for connections internal to a device, and CABLE for connections over external cabling. Support for other transport types may be added to future versions of the HAVi Stream Manager, or may be provided by proprietary extensions to the Stream Manager.

transmission format – identifies the transmission protocol used to send a stream type over a transport type. For IEC61883, the transmission format corresponds to the FMT and FDF fields in the Common Isochronous Packet (CIP) header. For CABLE, the transmission format identifies forms of signaling used on physical cables. HAVi assigns a 16-bit code to many of the signal formats commonly used by CE equipment, these are listed in Annex 11.12. For INTERNAL, transmission formats are not used.

plug – a resource, provided by a device and used to build a connection. A plug is, at least conceptually, the source or sink of the data carried by a connection. There are two main plug categories: *functional component plugs* and *device plugs*. A device plug is either a PCR or an external cable plug. Functional component plugs and device plugs are represented by FCM plugs and DCM plugs respectively. However it should be noted that FCM and DCM plugs are software abstractions and that when one says, for example, two FCM plugs are connected, this implies that the underlying functional component plugs are connected. This document also refers to *source plugs* and *sink plugs*, and *output plugs* and *input plugs*.

channel – a resource, provided by a transport mechanism (e.g., IEEE 1394) and used to build a connection.

3.7.4 Streams

The concept of *stream* as used by the Stream Manager is based on the isochronous data flow concept of IEC 61883 with two differences: The first difference is that a IEC 61883 data flow starts at a device source plug and ends at a device sink plug – while a stream typically starts at a functional component source plug, goes to a device source plug, then a device sink plug, and ends at a functional component sink plug. The second difference is that streams are typed, i.e. for each stream there is an associated *stream type* identifying data representation, bandwidth, and other attributes of the stream.

The following summarizes the properties of streams, channels and connections:

* a stream is associated with a single source and a set of connections and their channels

* device plugs can be *bound* to channels

* channels are either *fully bound* (bound to a source device plug and a sink device plug) or *partially bound* (bound to a source device plug only or to sink device plugs only)

* a channel can be bound to zero or one source device plugs and zero or more sink device plugs

* a sink device plug can be bound to at most one channel

* a source device plug can be bound to at most one channel

* a channel may have no source device plug, but no data will flow over the channel until a source device plug has been bound

3.7.5 Connections

3.7.5.1 Device Connections

A device connection is one of:

- An internal connection from a functional component source plug to a functional component sink plug.

- An attachment from a functional component source plug to a device source plug.

- An attachment from a device sink plug to a functional component sink plug.

The following rules apply to device connections.

- Functional component sink plugs can have at most one device connection.

- Functional component source plugs can have many device connections.

- Device source plugs can have at most one device connection.

- Device sink plugs can have many device connections.

Stream Managers establish and break device connections via the `Dcm::Connect` and `Dcm::Disconnect` APIs. Stream Managers shall request the DCM to connect even if the device connection already exists, this allows the DCM to "overlay" device connections and maintain information about the usage of device connections.

3.7.5.2 Internal Connections

Internal connections are a form of a device connection. Consequently:

- Functional component sink plugs can have at most one internal connection.

- Functional component source plugs can have many internal connections.

Stream Managers establish and break internal connections via `Dcm::Connect` and `Dcm::Disconnect`.

3.7.5.3 External Connections

An external connection will involve attaching functional component plugs to IEC 61883 plug control registers (transport type is `IEC61883`) or to external device plugs (transport type is `CABLE`). External device plugs are connected, typically, by physical cabling. It is not required that the HAVi Stream Manager be aware of the configuration of such cabling. If the Stream Manager is asked to establish a connection for which successful operation would require physical cabling (e.g., analog audio or video cables), the Stream Manager assumes the user has made the proper connections.

3.7.5.4 Global Connection Map

The Stream Manager is capable of constructing a map of all connections within the home network

established by HAVi applications. The Stream Manager does not guarantee that the map is built in one atomic operation.
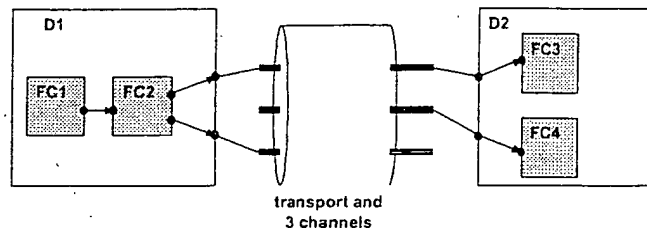
3.7.5.5   Connection Examples

Figure 20. Connection Diagram

The diagram above shows several examples of connections. Here *D1* and *D2* are devices and the *FCi* are their functional components, the small dark circles represent plugs. There are four connections:

- an internal connection between *FC1* and *FC2*

- an external connection between *FC2* and *FC3*

- an external connection originating at *FC2* (the channel is partially bound)

- an external connection terminating at *FC4* (the channel is partially bound)

Note that, for example, the connection between *FC2* and *FC3* involves two device connections (attachment of the *FC2* source plug to the *D1* source plug, attachment of the *D2* sink plug to the *FC3* sink plug) and the binding of the channel to the device source and sink plugs.

Note that for FCMs with source plugs and sink plugs, for example FC2, it is not guaranteed that such an FCM will or will not pass through the signal from a sink plug to a source plug.

3.7.6 Transport Types

Transport types are represented by 16-bit identifiers. The values assigned to the three transport types, CABLE, INTERNAL and IEC61883, are listed in 11.14.

3.7.7 Stream Types

Stream types are represented by a stream type identifier – a data structure containing an IEEE 1394 Vendor ID field and a type number field. The Vendor ID of value 0x0 is reserved for use with HAVi defined stream types. Stream types form a hierarchy, the HAVi stream types are arranged in

the hierarchy shown below (only the structure of the hierarchy and some representative stream types are shown, for a complete list of stream types see Annex 11.11):
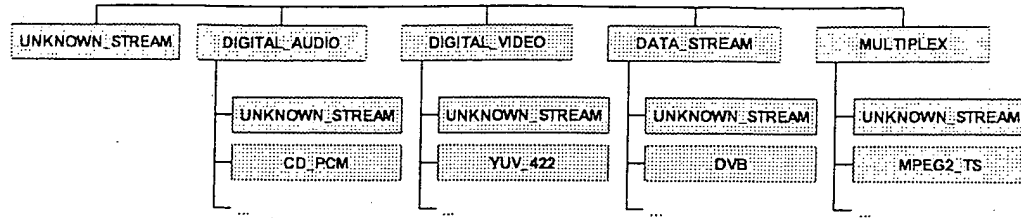

Figure 21. Stream Types

The stream types UNKNOWN_STREAM, DIGITAL_AUDIO__UNKNOWN_STREAM etc. are used for streams of unknown type. During connection establishment, the Stream Manager reduces stream type compatibility checks for such plugs. Instead it follows a "connect and test" procedure (i.e., it attempts to connect and then tests for success or failure).

HAVi reserves VendorId = 0 for stream types. Device manufacturers can define their own stream type hierarchy using their vendor id. Stream types with different vendor id will not match each other except for the UNKNOWN case (see 5.9.5.1 Stream Type Matching).

3.7.8 Plug Compatibility Checking

When the Stream Manager attempts to connect plugs it tests whether:

* transport type is compatible (i.e., a common transport type can be determined)

* directionality is compatible (i.e., output plug to input plug)

* stream type is compatible (i.e., the stream type of the source and sink plugs match as described in section 5.9.5.1)

* bandwidth is compatible (i.e., the required bandwidth of the source is less than or equal to the bandwidth which the sink is capable of consuming)

* transmission format is compatible (i.e., the transmission format of the source and sink plugs match as described in section 5.9.5.2)

If any of these tests fail then the connection cannot be established. The compatibility check about stream type, bandwidth and transmission format is not applied to the CABLE and INTERNAL connection.

3.7.9 Connection Restoration: Network Reset

After a network reset, the IEC 61883 procedures for restoring connections are invoked by the Stream Manager for each connection (of the IEC61883 transport type) that it has created. During connection restoration the Stream Manager builds a new list of connections for which it is responsible and may post ConnectionDropped events. This event is posted when:

* as a result of a network merge there is insufficient bandwidth available for a connection,

※ as a result of a network partition the source or sink of a stream has been lost.

Stream Managers shall complete IEC 61883 connection restoration during the first 1 second after network reset (this is the "isoch_resource_delay" period specified by IEEE 1394-1995). Also, the Stream Managers shall indicate to DCMs the device connections they wish removed by using Dcm::Disconnect.

3.7.10 Connection Restoration: Power Off

If a device running a Stream Manager powers down then during handling of the subsequent network reset the connections created by the Stream Manager will not be restored. However, since a Stream Manager only creates connections for local applications then the applications themselves will no longer be running so there should be no need for restoration.

3.7.11 Connection Dropped

A connection is dropped under the following circumstances:

※ the Stream Manager which created the connection is requested to drop the connection

※ the owner of the connection or an FCM for the source or sink leaves

※ the Stream Manager receives a PowerStateChanged event indicating loss of power for the source or sink

※ the source or the sink is no longer present after a network reset (as described above)

※ the Stream Manager fails to restore the connection after a network reset (as described above)

※ the Stream Manager detects a removal of an device connection (via the DCM DeviceConnectionDropped event) and determines that a connection the Stream Manager has established is no longer operable

3.7.12 Connection Changed

Stream Managers subscribe to TransmissionFormatChanged and StreamTypeChanged so that they can maintain a consistent view of the transmission format and stream type used by a connection. Stream Managers subscribe to BandwidthRequirementChanged, in order to be informed of changes (or attempted changes) in bandwidth associated with a connection. Finally Stream Managers subscribe to DeviceConnectionDropped and DeviceConnectionChanged in order to respond to changes in internal configuration of the source or sink of a connection.

3.7.13 Connection Establishment and Drop Order

Stream Manager establish connections of transport type IEC61883 by configuring from source to sink (i.e., first connect source attachment, set source stream type and transmission format, update the IEC61883 plug control registers, connect sink attachment, then set sink stream type and transmission format). Oppositely, Stream Manager should drop connections of transport type IEC61883 from sink to source.

If Stream Manager fails DCM/FCM API or set IEC61833 plug control register to establish a connection, Stream Manager does not need to restore these settings to the previous state, but shall restore the settings of isochronous resources and plug control registers according to IEC61883 CMP.

3.7.14   Connection Overlay

HAVi connections can be overlayed not only for IEC61883 transport type, but also for `CABLE` and `INTERNAL` transport types. Overlay of device connection part of HAVi connections are managed by DCMs and Stream Managers. DCM maintains a list of Stream Managers that have device connection on the DCM. Once a Stream Manager calls `Dcm::Connect`, the Stream Manager is memorized in the list in the DCM. The DCM will not maintain how many device connection is owned by each Stream Manager and each Stream Manager shall maintain the count by itself. (Note: Stream Managers should call the `Dcm::connect` for each time making overlay for reservation protection check reason.)

When an SE calls `StreamManager::Drop`, Stream Manager decrements the counter but does not call `Dcm::Disconnect` until no connection on the attach remains. On executing the last drop process, the Stream Manager calls `Dcm::Disconnect`, the DCM erases the Stream Manager from the list. In this way, overlay within a Stream Manager and overlay between Stream Managers are maintained.

In case of overlays, when the source and/or sink attachment already have the desired stream type the Stream Manager should not call `Dcm::SetStreamTypeId` again. Similarly, in case of overlays, when the source and/or sink DCM already have the desired transmission format the Stream Manager should not call `Dcm::SetTransmissionFormat` again.

3.8 Resource Manager

Applications in the network will typically use a set of FCMs to perform a task on behalf of one or more users. FCMs are called *device resources* in this context. Usually, also *network resources* are involved in resource management, since these serve to create useful collaborations in audio/video streaming between DCMs and FCMs in a HAVi network. 1394 bandwidth and channel numbers are such network resources. The Resource Manager only deals with device resources.

The Stream Manager handles network resources on behalf of clients. Connection and bandwidth management will typically be requested by applications after the involved device resources have been reserved for usage. In the following, resources stands for device resources, or FCMs.

Resource management serves to guide software elements competing for and using the set of resources in the network. Such software elements are called *clients* in this context. A group of device resources will be reserved by a client in an all-or-nothing fashion. The resource management system serves to ensure that clients that have reserved resources can rely on not being disturbed by other applications that (try to) use these resources. Potential clients that want to reserve resources, but have not yet done so are called contending clients or *contenders*.

There is a Resource Manager on each FAV and IAV device that hosts or can host at least one DCM. The **HAVi_Resource_Manager** SDD field of an IAV device reveals whether a Resource Manager is present.

3.8.1 Resource Reservation

Each Resource Manager offers methods to applications for reserving and releasing resources, as well arranging *scheduled actions*. A scheduled action is a reservation and usage of a set of resources during some future time period. Conflicts in schedules are detected and reported. The resource management system will perform a number of validations for scheduled actions at scheduling time.

The resource management model supports a reservation mechanism. Reservation is used to protect against *control commands* ("write access"), changing the state of a resource. Reservation is not needed for *observation commands* ("read access"). An FCM is not obliged to support reservations if it only has methods that do not change its state. In this case, FCM methods and events related to resource management need not be implemented. Other FCMs shall support reservation facilities.

In general, applications can subscribe to resource events. Such event subscriptions are considered not to change the state of the resource, and therefore belong to the observation category. The FCMs and Resource Managers offer a number of events that can be subscribed to by interested applications to learn about the status of resources with respect to reservations.

Two client roles are distinguished and are relevant in *negotiations* for resources (discussed later):

- *User* – A client that is able to take part in a resource negotiation with a contender. Negotiation will typically involve the human user of the client (to give him or her the opportunity to accept or reject a negotiation). However, it is not required that a human user be involved for a client to take this role.

- *System* – A client that is not able to take part in a resource negotiation with a contender. This will typically be the case if no human user is involved with the client.

A limitation of a system contender is that it can or should not preempt resources from user clients that reject its preemption request. Resource Managers performing a scheduled action (Action Schedulers) are such system contenders. An application is free to act as a user or system contender, although it should choose the correct role in a cooperative situation. User contenders can always preempt resources from any client.

Note that, for instance, an emergency application would typically be a user contender. Although it may acquire resources by direct preemption, it may negotiate first to learn whether a current client has an even higher priority.

A DDI Controller may control a DCM through the DDI protocol. The DDI Controller is responsible for reserving any FCMs associated with the DCM if it requires exclusive access to the related device or some of its functional components. See section 3.5.1.8 Resource Management.

The resource management model relies on the following principles:

- Clients are cooperative, i.e., they shall respect the reservation mechanism and their user (or system) role.

- Before using control commands, an application will normally reserve the needed resources. It shall release the resources when they are no longer needed.

- With due restrictions, a contender may negotiate through a Resource Manager with other clients to take over (preempt) their reservation of resources. It may also decide to preempt resources without negotiation. This process is described later.

75

The next figure shows an example of the interaction between clients in a network with FAV or IAV TV and VCR devices. (In this example, only one resource is reserved. In general, a Resource Manager will attempt to reserve a group of resources requested by a contender.)
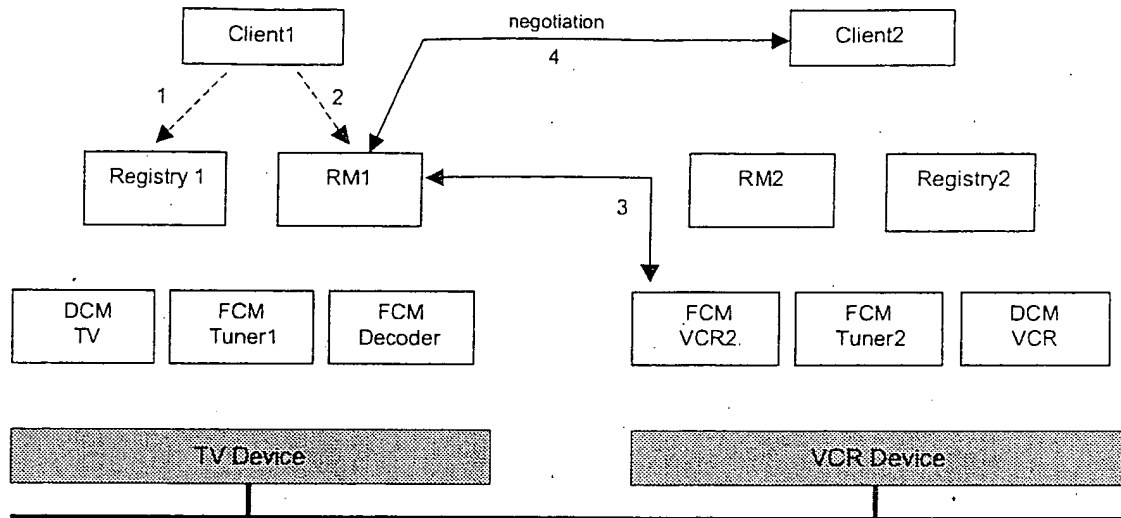

Figure 22. FCM Reservation

Client1 first queries its local registry for VCR resources present in the network (arrow 1) and then calls a (local) Resource Manager (*RM1*) in order to reserve the VCR FCM (arrow 2). Next, *RM1* attempts to reserve *VCR2* (arrow 3).

Assume *VCR2* is currently reserved by *Client2*. Before attempting to reserve it, *Client1* may negotiate through a Resource Manager to preempt the resource from *Client2*. It will then send a negotiation request to, say, *RM1* (arrow 2). *RM1* will send a preemption request to *Client2* and await a preemption reply from it (arrow 4). An acceptance or rejection is forwarded by *RM1* to *Client1*. Depending on the answer, *Client1* may decide to preempt *VCR2* (even if *Client2* rejected).

There are various types of resources. Some will provide function for producing AV contents like Tuner. Some other will provide both function for producing or recording AV contents like VCR or AVDisc. Yet other will provide audio visual interface to the users like Display or Amplifier. HAVi does not specify which resources to be reserved, but clients are recommended to cooperate respecting following suggestion; As for FCMs that provide audio visual interface like Display or Amplifier FCM, the basic usage may be "free to use" model, that means clients will not reserve FCM without special purposes or reasons. When clients reserve these FCMs, the clients have to take into account the side effect of the reservation that might cause inconvenience to the user.

3.8.2 Resource Sharing

Device resources can be reserved by an application for exclusive use, but can also be shared by more than one application. The resource type determines which sharing options are available to applications.

Any application may change the state of a device resource as long as the resource has not been reserved. However, resources shall protect applications from unauthorized access by other applications in case the resource is properly claimed. A resource shall verify for each command whether the invoker is entitled to perform the command. (It shall check the SEID of the invoker

with its locally registered primary and secondary SEIDs for control commands.)

A resource may be shared by more than one client. A DCM provider can choose the extent of sharing that is supported for each FCM, and the restrictions that apply to sharing clients with respect to each other. Two types of access rights are distinguished:

▓ *Primary access* – Full control of the resource without any restriction. There is at most one primary user per resource.

▓ *Secondary access* – Limited control of the resource. The primary client and other secondary clients should never be hampered in their access rights by a secondary client.

The number of secondary clients allowed is determined by the DCM manufacturer. If a primary client releases the resource, a secondary client (if any) will not automatically become the primary client. It may, if desired, attempt to reserve the resource as a primary client. If successful, it will become a primary client, and loses its secondary client position.

As an example, consider a tuner FCM able to operate in one of several multiplexes, each of which supports a number of channels. The primary client can change the multiplex of the tuner, but a secondary client may not. The secondary client can only tune to a channel that happens to be in the selected multiplex.

Note that any application can use observation commands of a resource at any time. It need not reserve the resource for such commands. Also note that user and system clients may accept to operate with either primary or secondary access rights. However, negotiation and a subsequent preemption (discussed in the next section) only apply for primary access rights.

3.8.3 Resource Negotiation and Preemption

A resource is normally used by an application between the moments it chooses to reserve and release the resource. However, there are means for competing applications to hand over resource access rights.

Reserving and releasing resources is always done by a Resource Manager selected by the contender using the methods `ResourceManager::Reserve` and `ResourceManager::Release`. Only Resource Managers are allowed to directly reserve and release resources through `Fcm::Reserve` and `Fcm::Release`.

A user contender can always become the new client of a group of resources by preempting them from their current clients. However, in normal circumstances, a contender will first try a so-called *non-intrusive reservation*. This is a reservation attempt that does not involve any current client. If all resources in the requested group can be acquired in this way, they will be acquired; otherwise, none of them will.

A contender can negotiate with primary clients through a Resource Manager in order to learn whether they are willing to give up their resource reservations. The method `ResourceManager::Negotiate` is used by the contender. The Resource Manager will then invoke the method `<Client>::PreemptionRequest` of all primary clients to forward the request. Each client should respond within a specified timeout with an acceptance or rejection of the request, and an information string. (`PreemptionRequest` should be implemented by applications that take on the user role). Cooperative controllers are advised to follow the following rules:

- If all clients accept the preemption request conveyed in the negotiation, the contender may preempt their resources.

- A system contender shall not preempt resources from clients that have rejected the preemption request.

- To all clients that have received a preemption request, the contender shall send a withdrawal notification if it decides not to preempt their resources (through `ResourceManager::Negotiate`).

So, normally negotiation precedes preemption. The client should respond within a specified negotiation timeout. This negotiation may involve a person behind a current primary user client to agree or disagree with a preemption. However, even if a client disagrees, a user contender can directly preempt its resources. This prevents any application in the network from monopolizing resources. If a negotiation times out, the contender may decide not to preempt the resources of that client, although this is up to its own discretion. This scheme is thought to suffice for home networks, where applications should cooperate in the usage of resources.

Negotiation, preemption, and reservation are always done for a resource group, although a client can choose any subset of resources to reserve or release, even after it has already reserved other resources.

The following scheme describes situations where preemption is recommended or allowed after negotiations between a contender and a set of clients.

preemption after negotiation	user contender	system contender
current user client	upon accepted negotiated reservation requests (preemption always allowed)	upon accepted negotiated reservation requests (else preemption not allowed)
current system client	negotiated reservation requests not accepted (preemption always allowed)	negotiated reservation requests not accepted (preemption not allowed)

For system contenders, preemption is *only* allowed if the current client is a user client, and has accepted the request. For system clients, the resource management system shall always reject reservation requests from system contenders. A system client shall never directly preempt resources, or do so without the required negotiation acceptance. (Note that a system contender negotiating with another system client will fail, because system clients are assumed not to support negotiations.)

If a non-intrusive reservation fails, the contender may analyze why it failed. This can be done by processing the results of the reservation command returned by the Resource Manager. Any application can use `Fcm::GetReservationStatus` to learn about various reservation properties that hold for the resource. In general, applications should subscribe to `ReserveIndication` and `ReleaseIndication` events if they need to be aware of reservations and releases, including those related to resources they have reserved themselves.

For a group of resources, it can be specified whether a non-intrusive or preemption approach should be used in a reservation. However, only if all resources can be reserved, will they indeed be reserved. Otherwise, none will be reserved.

78

Due to other reservations, missing resources, or network partitions, a reservation command may fail for one or more resources. After invoking ResourceManager::Reserve and ResourceManager::Negotiate, a contender should always consult the status or negotiation record for each individual resource.

Application requirements. At any time during a reservation, a primary client can receive one or more negotiation messages from Resource Managers. A client shall accept and process them immediately. The client should respond within the specified timeout period. It should receive a follow-up message from a Resource Manager if the preemption request is withdrawn. It is not necessary that a current client releases the requested resources; this will be done by the resource management system. The client can learn about the actual preemption by subscribing to the ReserveIndication event.

Example. Below is a scenario that sketches how a user and a system client would interact with the resource management system (abstracted by RM) and the resource (R).

In this scenario, a + after a method denotes an accept response on the command. A + or - on a dashed arrow denotes an accept or reject response on the earlier command (without + or -). NIR stands for non-intrusive reservation, GRS stands for GetReservationStatus (an FCM method).

The scenario illustrates that a user client may be using a resource, possibly being unaware of, say, an Action Scheduler, which operates as a system client. An Action Scheduler shall always attempt a non-intrusive reservation (NIR) first. If this fails, it will typically negotiate for the resources it needs. The user client is informed of the scheduled action that was planned. In this case, the user chooses to accept a preemption of the resource, so that the scheduled action can succeed.
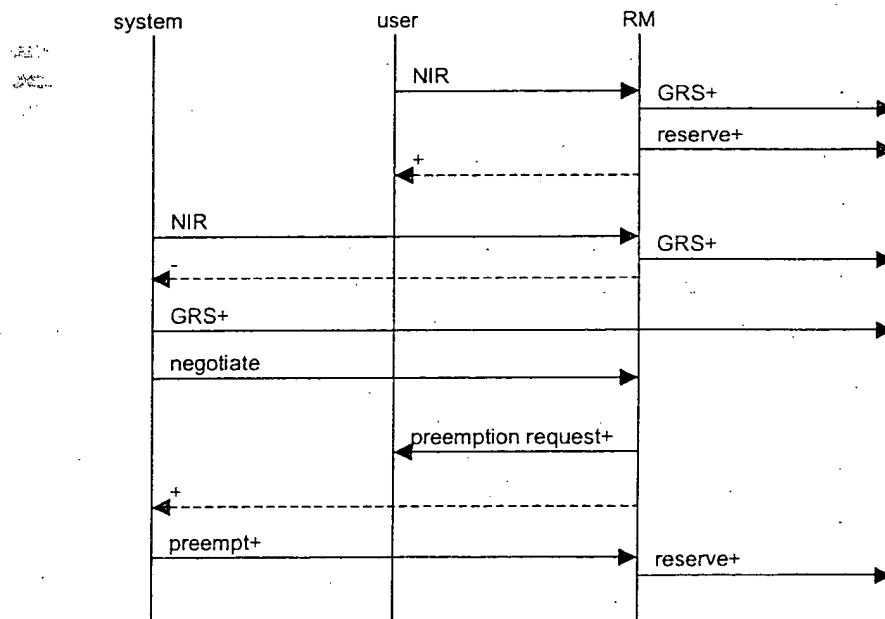

Figure 23. Reservation Protocol

3.8.4 Scheduled Action Management

The goal of Scheduled Action Management is to allow and secure Scheduled Action programming by the system. That is:

▓ to ensure that at *starting time*, all the necessary resources (FCM/DCM, bandwidth, …) are present and thus can be reserved

▓ to check at *scheduling time* the feasibility of the action at starting time i.e. to check the feasibility of the commands planned on the (device) resources and the planned inter-connections (network resources) between them

▓ to perform at *starting time* all the planned reservations and actions

Management of a Scheduled Action involves, apart from the Resource Manager, the following components:

▓ An application that specifies the scheduled action (the *invoking application*). The invoking application is only involved in the ordering of a Scheduled Action. If it disappears afterwards, the execution of the Scheduled Action is not affected.

▓ One or more FCMs or DCMs that are needed to successfully complete the required scheduled action.

▓ (Optionally) a *trigger* (application or FCM) that notifies the Resource Manager to start the action.

▓ (Optionally) an application that is involved in the scheduled action in the sense that it takes control over the involved resources during action time (*control application*).

Transactions (reservations, etc.) between FCMs, DCMs and applications such as described above are performed via a Resource Manager in an architectural model discussed below. First an overview of the data involved in a Scheduled Action will be given.

3.8.4.1   Scheduled Action Data

A Scheduled Action is defined by the following information:
* SEID of Scheduled Action controller (optional)
* SEID of Scheduled Action trigger (optional)
* Start Commands list (in a strict sequential order)
* Stop Commands list (in a strict sequential order)
* Connection list (in a strict sequential order)
* Start and Stop Time information
* Involved Resources HUID List
* User information (optional)

Scheduled Action controller: the application that will be awoken when the Scheduled Action is executed. This application may control the resources. If the field is absent, the Scheduled Action will not be controlled during execution but will execute autonomously.

Scheduled Action trigger: this field indicates the trigger (e.g., FCM or DCM) that will generate the actual notification for starting/stopping the execution of the Scheduled Action. The start/stop times will not be used in this case to actually start/stop the Scheduled Action's execution, but will be used to coordinate the reservation of resources with other schedules.

Start / Stop Commands: these are ordinary HAVi commands (defined by the APIs). They must be listed in order of execution and it has to be specified which resource must execute them. This sequential order is needed to ensure that a command cannot be sent to a resource without being sure

that the previous command has been successfully executed by the relevant resource. Commands can be either FCM commands or DCM commands.

Connection List: these indicate the network resource allocations needed for the Scheduled Action. From this list, the FCM plug connections that ultimately will be executed by the Stream Manager, will be constructed.

Start and StopTime Info: time information needed for scheduling an action (including date, start and stop time and periodicity data).

Involved Resources HUID List: this list represents the resources that are involved in the Scheduled Action.

User Information: this optional field can contain a description of the Scheduled Action that provides useful information in case a scheduled action that was originally accepted can no longer take place.
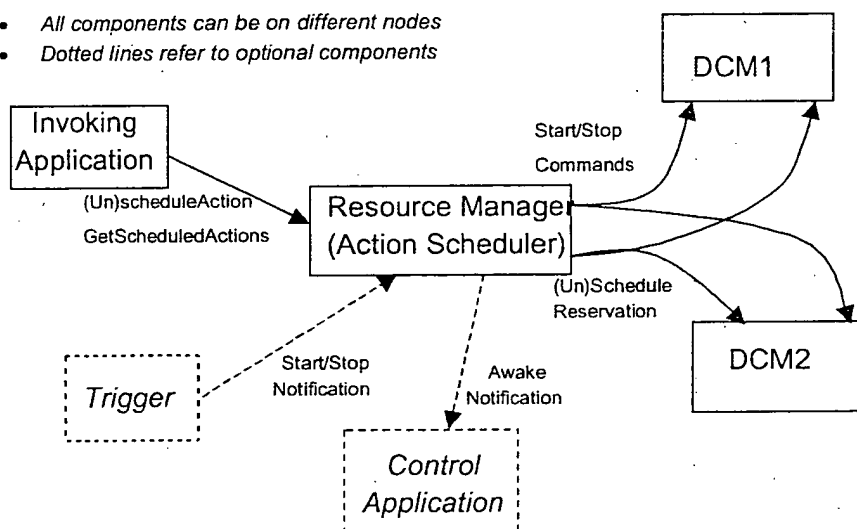
3.8.4.2   Scheduled Action Model

Figure 24. Resource Manager and Scheduled Actions

Figure 24 provides an overview of the involved software elements in a Scheduled Action. A description follows.

3.8.4.2.1   Scheduled Action

The invoking application passes its Scheduled Action (including Scheduled Action parameters as described above) to a Resource Manager of its own choice. The invoking application might choose its local Resource Manager, or take into account that the chosen Resource Manager can best be at the same node as one of the involved DCMs or trigger or control application (to decrease the chance of problems due to network changes). The Resource Manager chosen will be called *Action Scheduler* in the sequel.

The Action Scheduler persistently stores the Scheduled Action from the time the Scheduled Action

is invoked until the time it is completed and no longer needed. Per Scheduled Action only one Action Scheduler is involved. In case of network changes (e.g., network re-join after partition), it will take care that its Scheduled Actions are recovered (see below: Network Changes).

The Action Scheduler will assign a unique (local) index to the Scheduled Action for future reference. After acceptance of the Scheduled Action this index is eventually returned to the invoking application. A Resource Manager shall reuse these indexes as little as possible.

The invoking application is responsible for gathering all information about the involved FCM HUIDs, time info, connection list, etc. The model does not assume that all involved FCMs in a Scheduled Action are reservable FCMs: also non-reservable FCMs shall be accepted for Scheduled Actions.

Data Passed to Action Scheduler
SEID of control application(optional)
SEID of trigger (optional)
Operation code for awake notification message
Start Commands list
Stop Commands list
Connection list
Start time/date
Stop time/date
Periodicity
Involved FCMs HUIDs
User info (optional)

Below is the basic structure of a start or stop commands list (see also section 5.10.2). The format of the "command to be executed" is exactly the same as the (content of the) message that will take care of its execution. The Action Scheduler will not interpret the command itself (and considers it as a sequence of bytes).

Start or Stop Command List	
Command to be executed	HUID of the FCM or DCM that must execute it
...	...

The structure of the connection list allows the Action Scheduler to defer from it the eventual Stream Manager commands (FlowTo). The connection list looks as follows (see also API section):

Connection List		
Source FCM Plug.	Sink FCM Plug	Stream Type
...	...	...

The invoking application might use a slightly different time for "start" then the one that was specified by the user, in order to take into account that the execution of all the actions can readily be done beforehand. The application can make use of GetWorstCaseStartupTime of the involved FCMs for calculating a worst case start-up time.

3.8.4.2.2   Schedule Reservation and DCM Checking

The Action Scheduler contacts the DCMs of each involved resource to distribute *Restricted Scheduled Actions*. Restricted Scheduled Actions are restricted versions of an original Scheduled Action that contain only the data that concern the resources local to each DCM. By this means, the FCMs local to the DCM can be checked (if the resource is free at scheduled time) and be aware of the actions (commands and connections) to perform at start or stop signal.

Data Passed to Involved DCMs
Start Commands list
Stop Commands list
Connection list
Start time/date
Stop time/date
Periodicity
Involved FCM HUIDs
User info (optional)

The start and stop command lists consist of a subset of the original Scheduled Action lists. Only those commands are gathered that belong to the particular involved DCM (the *target DCM*). The connection list is also constructed from the original Scheduled Action list. Only those entries are copied that refer to an FCM "inside" the target DCM. The same holds for the list of involved FCMs.

In this way each involved DCM is able to build up an internal agenda for reservations of resources. The internal agenda allows a DCM to check for schedule overlaps. For doing this the DCM has to take into account the start/stop time/date attributes and also the periodicity attribute. In case a DCM detects a schedule overlap, it checks whether it (and its FCMs) will be able to execute simultaneously the intended actions (start/stop commands and connections) during the (overlapping) time period.

A DCM shall at least check whether there is a schedule overlap. This can require checking of several schedules if periodicity was specified. Further checking of the involved actions is highly recommended because otherwise possible problems will only become apparent at starting time of the Scheduled Action.

Further notes on checking:

For connections from an internal FCM to an external FCM (internal/external with respect to the target DCM), only the DCM internal part has to be checked. The other parts are checked by another DCM and the bandwidth availability is checked separately (see section 3.8.4.2.3).

The Scheduled Action reservation of involved resources can lead to blocking of *dependent* resources. Dependent resources are separate resources (FCMs) which can no longer be used freely because of, e.g., already established connections. This means that different Scheduled Actions involving different FCMs (within the same DCM) can still be conflicting. The blocking of dependent resources is "invisible" to resource management: a DCM is responsible for taking into account the possibly resulting blocking of dependent resources when accepting a scheduled reservation.

A consequence of the above is that DCM checking can be rather complex. However, DCMs can also choose to do things rather simply by allowing only restricted reservations (e.g., every reservation reserves the complete DCM). The "checking complexity" that a DCM wants to deal with also determines how much information of the Restricted Scheduled Action the DCM has to store.

After checking, the DCM returns to the Action Scheduler whether it accepted (its part of) the Scheduled Action.

3.8.4.2.3 Bandwidth Checks

If all (local) DCM checks are OK, the Action Scheduler will check whether the needed network resources will be available at starting time. This check has to be performed only for connections between FCMs on different DCMs. The network resources to be checked for availability are bandwidth and channel numbers. Since these are global resources, the Action Scheduler that is responsible for a new schedule has to inquire all other Action Schedulers about already programmed scheduled actions that will overlap in time with the new schedule.

The checking, which shall be done at scheduling time, can be conceived as taking place in two steps:

1. The Action Scheduler collects all planned connections corresponding to existing schedules in the network that overlap in time with the new schedule. It uses the `ResourceManager::GetScheduledConnections` API for this purpose. In this way the Action Scheduler can build up a bandwidth table of needed bandwidth and channels during the period of the new schedule. (The actual bandwidth calculation method is described in section 5.10.5.)

2. The Action Scheduler then compares the network capacity (total available bandwidth and total available number of channels) with the information in the table and thus checks if there will be enough bandwidth (and channel numbers) at execution time of the new schedule.

Note that to perform the bandwidth related calculations the Action Scheduler may utilize already available Stream Manager facilities. Since all calls to the Stream Manager are local and the Action Scheduler and the Stream Manager will be from the same vendor, the Stream Manager services needed to do the calculations are proprietary and not part of this document.

3.8.4.2.4 Usage of Timers and Triggers

If the above bandwidth checking or one of the previous DCM checks failed, the Action Scheduler will cancel the Scheduled Action. The local scheduled reservations on the DCMs will be undone via `Dcm::UnscheduleReservation`.

If all checks are okay and no trigger is specified, the Action Scheduler will use a timer in order to get notified when the start/stop times happen. When no trigger is specified, the Action Scheduler will set the start time according to the next valid value of the timer.

The Action Scheduler may use a Clock FCM for setting up the timer start/stop times, however, the Action Scheduler should be able to complete the Scheduled Action (in a proprietary manner) also when no Clock FCM is available. For a timer facility used by the Action Scheduler (Clock FCM or proprietary) a worst case accuracy of one minute is required.

In case a trigger was specified, the possibly needed subscription is not dealt with by the Action Scheduler. The application is responsible for setting up the trigger notifications. If the trigger start notification is never sent, for one reason or the other, the Scheduled Action will remain existing (waiting for the notification). It is up to the user (application) to remove the Scheduled Action. The same holds if the trigger stop notification is never sent. The reservations done will remain until the Scheduled Action is removed.

3.8.4.2.5 Executing the Scheduled Action

When the Action Scheduler is notified (either by a timer or by the trigger) that the Scheduled Action should be executed, it first executes all reservations of the involved FCMs (via `ResourceManager::Reserve`). This is a non-intrusive reservation with client role of "system"

and requesting primary access rights. If the reservations fails the Action Scheduler will start negotiating as a cooperative controller (via `ResourceManager::Negotiate`, see section 3.8.3). Then it uses the connection list to establish `FlowTo` connections via the Stream Manager. Before establishing connections, the Action Scheduler turns on the power state of the DCMs, which are needed to establish the connection, to avoid connection failure. Each `FlowTo` should use "any" values for `ConnectionHint` except for stream type. Initially a `FlowTo` will be attempted with `dynamicBw` equal to `False`. If this fails, a `FlowTo` with `dynamicBw` equal to `True` will be attempted. After connections have been established, the Action Scheduler then executes all start commands. All these are HAVi commands of the format described in 3.2.3.4. The connections and commands are executed in the order initially given by the (invoking) application. In case the reservation fails, or a command or connection does not return `SUCCESS`, the Action Scheduler will abort the complete Scheduled Action and generate an `ErroneousScheduledAction` event.

In case a control application was initially specified, this application is awoken by the Action Scheduler via `<Client>::AwakeNotification`. Note that such application awaking is optional, and is not required to complete a Scheduled Action. In this case, the above reservations are not performed by the Action Scheduler but by the control application itself. This is done in order not to disturb the owner relationship of the resources. For the same reason the connections and command executions will be executed by the control application. This means that in case a control application was initially specified, the start/stop commands etc. of a Scheduled Action are only used for checking. This also means that the control application has to deal with failing reservations, etc.

If everything has been started, the Action Scheduler will use the specified method (timer or trigger) for stopping the Scheduled Action. When no trigger is specified, the Action Scheduler will set the stop time according to the next valid value of the timer.

3.8.4.2.6 Ending the Scheduled Action

When the Action Scheduler is notified (either by timer or trigger) that the Scheduled Action should be stopped, it executes all stop commands, breaks all connections (via `StreamManager::Drop`) and releases the resources (via `ResourceManager::Release`). Again, these are ordinary, existing HAVi commands. The commands are executed in the order initially given by the (invoking) application. If the Scheduled Action was not used with "periodicity", it will be removed from the list of Scheduled Actions in the Action Scheduler list and via `Dcm::UnscheduleReservation` also from the internal agendas of the involved DCMs. When the Scheduled Action is used with "periodicity" and no trigger is specified, the Action Scheduler will set the start time according to the next valid value of the timer. When the Scheduled Action is used with "periodicity" and a trigger is specified, the Action Scheduler will do nothing.

In case a control application was initially specified, the above actions are not executed: if the application did not already finish it will continue as in "normal" (unscheduled) operation. The removal of the Scheduled Action however will take place.

3.8.4.3 Query and Modification of Scheduled Actions

An application can consult the database of Scheduled Actions maintained by the Action Schedulers. It can use a facility to get an overview of all Scheduled Actions stored in a given Action Scheduler (`ResourceManager::GetLocalScheduledActions`), and can inspect individual Scheduled Actions in detail (`ResourceManager::GetScheduledActionData`).

Applications can use these facilities for removing no longer needed Scheduled Actions (e.g. periodic ones) via `ResourceManager::UnscheduleAction`. The Action Scheduler will then cancel the Scheduled Action and the scheduled reservations on the involved DCMs will be undone

via Dcm::UnscheduleReservation. A change in a Scheduled Action can only be performed by first removing it and then scheduling it again (with appropriate changes).

3.8.4.4 *Network Changes*

Here is the model showing how the Resource Manager takes into account changes in the HAVi network. After network change, all scheduled actions are rechecked. This may result in some warning messages, e.g., while changing the cabling.

The Action Scheduler keeps track ("watch-on") of all involved DCMs and (if specified) the trigger and control application. If one of the DCMs disappears due to a network change, the Action Scheduler will react as in case of an UnscheduleAction, although it will not remove its own copy of the (complete) Scheduled Action. Instead, it shall 'remember' that it is invalid. However, the Scheduled Action will not be executed as long as the missing resources are not back on the network. Note that the ResourceManager::UnscheduleAction involves Dcm::UnscheduleReservation, which might not be feasible for all the involved DCMs since some may have disappeared. The Action Scheduler shall generate an InvalidScheduledAction event with the user info of the Scheduled Action as parameter. In case the specified trigger or control application disappears, the AbortedScheduledAction event is generated (ErroneousScheduledAction if the Scheduled Action is executing) and the Scheduled Action is completely removed from the system (as by ResourceManager::UnscheduleAction). It is therefore recommended that a Resource Manager is chosen on the same node as at least the controller application.

- For each Scheduled Action, each DCM keeps track ("watch-on") of the associated Resource Manager (Action Scheduler). If the Action Scheduler disappears due to a network change, the DCM will remove the (Restricted) Scheduled Action from its internal agenda and generate an InvalidScheduledAction event with user info as parameter. Note that this can also happen when the Scheduled Action is already in execution.

- If a new DCM is installed on a node, the Action Scheduler is notified of the installation of the FCMs via NewSoftwareElement events, and it will check if an 'invalid' Scheduled Action is associated with them (by checking the HUIDs). If all resources are available again, the Action Scheduler will try to restart the Scheduled Action mechanism. If the restart fails (due to new Scheduled Actions in the mean time), an AbortedScheduledAction event is generated.

- After a network change the available bandwidth may decrease, and as a result some scheduled actions may no longer be able to proceed. Each Action Scheduler that has stored Scheduled Actions is responsible for rechecking the available bandwidth after a network reset event is received. An Action Scheduler that detects a lack of bandwidth generates an InvalidScheduledAction event to inform the user and makes the Scheduled Action invalid. If bandwidth becomes available again the Scheduled Action is re-installed.

- Even if the resources are still not available at start time, the Scheduled Action is not removed: it must be possible to launch the Scheduled Action even if it is delayed. Only if the Scheduled Action is still not feasible at stop time, and only if it is not a triggered Scheduled Action, the copy of the Scheduled Action in the Action Scheduler is removed and an event AbortedScheduledAction is generated to inform the user.

uploaded DCM will work on a variety of FAV devices all with potentially different hardware architectures. To achieve this, uploaded DCMs are implemented in Java bytecode. The Java runtime environment on FAV devices supports the instantiation and execution of uploaded DCMs.

Once loaded and running within an FAV device, the DCM communicates with the LAV and BAV devices in the same manner as described above in section 2.7.1.

The efficiency of Level 2 interoperability appears when one considers resources needed to access device functionality. Level 2 allows a device to be controlled via an uploaded DCM that presents all the capabilities offered by the device. Whereas to achieve similar functionality in Level 1, this DCM would have to be embedded somewhere in the network. For example when a new device is added to a network, Level 1 requires that at least one other device contains a DCM suitable for the new device. In comparison, Level 2 only requires that one device provide a runtime environment for the uploaded DCM obtained from the new device.

The concept of uploading and executing bytecode also provides the possibility for applications called *havlets*. Havlets may be device specific, for example a device manufacturer can provide the user a way to control special features of a device without the need for standardizing all the features in HAVi. Havlets can be uploaded and installed by each FAV device on the network. Havlet uploading can be supplied by DCMs and Application Modules and can offer interaction with the user via Level 2 UI. Display-capable FAVs will allow a user to upload and execute the havlet of any DCM or Application Module, providing a havlet, in the home network.

2.8 Versioning

Each HAVi component must support the HAVi version control API. HAVi version control is intended to maintain interoperability of HAVi components as the specification evolves. Version control for individual manufacturer's products is outside of the scope of this API.

Versions are represented by major and minor numbers in the form `major.minor`. For a given release of the specification, every system component will be of the same version as that of the HAVi specification – regardless of whether that component's API was modified in the latest release. This simplifies the situation in which a given component's APIs are built up from different groups of APIs. The minor version number is intended to indicate small refinements of the HAVi specification. The intent of the major version number is to reflect important functional improvements in the overall HAVi architecture.

Given that this specification is Version 1.1, all HAVi components are currently defined to be at Version 1.1.

As the HAVi specification evolves, it is intended that no APIs will ever be updated or removed (see section 5.1.7). All changes will be achieved by adding new APIs. This ensures that older components can always request services from newer components successfully.

All system components on a device shall return a unified version number in response to `GetVersion` API calls, and the value shall be identical to **HAVi_Message_Version** in the SDD of the device. System components shall verify the version number of their peer system components on other devices, e.g., by reading the **HAVi_Message_Version** value from the devices' SDD (see section 9.10.3). Each system component shall operate at the highest common version number among all peer system components in the network.

The rules for version control of message passing protocols is somewhat different. Please refer to the Messaging System section 3.2.1.2.7 for details.

Client applications are encouraged to interoperate with older software element versions, though this is not explicitly required. However system elements, DCMs and FCMs as clients shall interoperate with all older software element versions. This is intended to ensure that interoperability will be achieved for the life of the HAVi specification. Note that all software elements support a version number retrieval method (see section 5.13).

2.9 Security

All software elements can in principle send messages and events to each other without any restriction. However to avoid that applications send, whether accidentally or deliberately, messages or events to system components that were intended to be used only by other system components, a protection mechanism is needed.

HAVi specifies, for each defined HAVi message and event, the kinds of software element that are allowed to use it. The protection mechanism simply implies that a system component will check whether the message sender (or event poster) is allowed to send this message (or event). This check is based on an inspection of the SEID of the message sender (event poster).

Protection of a device (and the home network) from hostile or flawed applications is the responsibility of the vendor of the device. Such protection is particularly crucial for FAV devices since HAVi specifies an open programming environment for FAVs and arbitrary bytecode applications may be introduced to FAVs (e.g., via Web download, broadcast download, or installation from hard media).

2.9.1 Access Levels

HAVi uses a two-level protection scheme. When a software element is created it is assigned an access level which is one of trusted or untrusted. When one software element sends a request to another software element the receiver decides whether or not to honor the request by examining the access level of the requester (and optionally other information associated with the request).

1) System software elements should be thoroughly tested by vendors. They are assigned the trusted access level and operate with the greatest set of privileges.
2) For non-system software elements that are pre-installed on an IAV or FAV, the vendor shall test the software element for HAVi compliance. Provided there are no failures, the software element can be assigned the trusted level.
3) For software elements that are dynamically installed on an IAV or FAV through an installation mechanism that is proprietary and not publicly exposed, the vendor shall implement a proprietary verification mechanism and only assign the trusted level to software elements obtained from secure sources.
4) For software elements that are dynamically installed on an FAV through a public installation mechanism, the vendor shall implement the signature verification mechanism defined by HAVi and only assign the trusted level to software elements obtained from correctly signed sources.
5) If an IAV or FAV has a proprietary mechanism for installing patches to system elements or replacing system elements, then it is recommended that the vendor implement a proprietary verification mechanism and only allow patching or replacement of system software elements from secure sources.

All DCMs and FCMs shall be trusted. Therefore untrusted DCM code units shall not be installed on FAVs or IAVs.

Since HAVi does not specify a dynamic acquisition and installation mechanism for Application

3.9 Application Modules

Application Modules are conceptually similar to DCMs. They provide an interface to a service that is usually provided by software only. An Application Module is a normal HAVi object in the sense that it communicates via the HAVi Messaging System. Like DCMs, Application Modules have HAVi Unique IDs. The HUID allows other applications to find the Application Module after partial system unavailability.

Since applications typically provide proprietary functionality, the standardized part of the API is small. It consists of basic identification (HUID) and visual representation (icon) and a means for providing a user interface. The user interface may be provided in two ways, corresponding to the two levels of interoperability: via DDI or via an uploadable havlet (similar to a havlet for a DCM). For an Application Module it is optional whether to support one or both ways of providing a user interface; what is supported is indicated by the ATT_GUI_REQ attribute in the Registry.

Application Modules can be provided by third-party application writers. Application Modules written in Java may be handled by arbitrary FAVs, they use the same format as DCM code units. The way IAVs handle Application Modules is proprietary. Also the way an FAV finds and installs third-party Application Modules (via a disk, Internet or any other means) is proprietary to the FAV node.

An FAV or IAV device is responsible for assigning the HUID to any Application Module running on the device. For Application Modules, two different HUID schemes provide different levels of persistency.

3.10 Code Unit Authentication

3.10.1 Outline of digital signature algorithm

All uploadable DCM code units shall be signed. Havlet code units may also be signed. HAVi specifies a public digital signature algorithm for these two kinds of code units. However, it may or may not be applied for Application Module code units, since verification for Application Module is vendor-dependent.

In HAVi Authentication, a public key system based on Rivest Shamir Adleman (RSA) is used. Also, Secure Hash Algorithm revision 1 (SHA-1) is used to digest the messages.

The HAVi Certification Authority (HCA) defines and keeps a unique pair of private key (never disclosed) and public key. The HAVi Certification Authority (HCA) also issues several pairs of private key and public key to each vendor on request, with the HCA's digital signature for the vendor unique public key. In this way, each vendor will have flexibility to be able to generate final signature to their products (DCM code units and havlet code units).

* In the HAVi specification of this version, HAVi public key is a 2048 bits key defined by HAVi, and statically stored in each FAV.
* vendor unique public key is one of 1024, 1536 or 2048 bits key assigned by the HCA, and included in a certificate
* HAVi uses RSA algorithm for digital signature compliant to PKCS#1 v2.0 [19] (See section 3.10.1.1).
* HAVi uses well-known 160 bits SHA-1 Message Digest [20].

A HAVi-signed code unit includes some specific files necessary for authentication in the JAR file. When an FAV is to install a HAVi-signed code unit, it has to verify the code unit with these files in advance to install the software element.

Uploadable DCM code units and havlet code units may be located in different locations with different security risks. HAVi therefore distinguishes between "External code units" and "Embedded code units". "External code units" are code units originating from device external locations like a Web site or an unsecured packaged-medium, e.g. floppy. "Embedded code units" are code units located in the SDD of a BAV, in a DCM/Application module (for havlets), or in secure storage (e.g. resident storage like NV-RAM or HDD in FAV or BAV, or proprietary secured memory card).

Thus, each vendor can request to obtain one or more pairs of vendor-unique keys from the HCA. These keys may be different in its size or validity period. These keys may be used for only External code units, or for only Embedded code units, or both.

The following figure shows the outline structure of certificates/signatures.
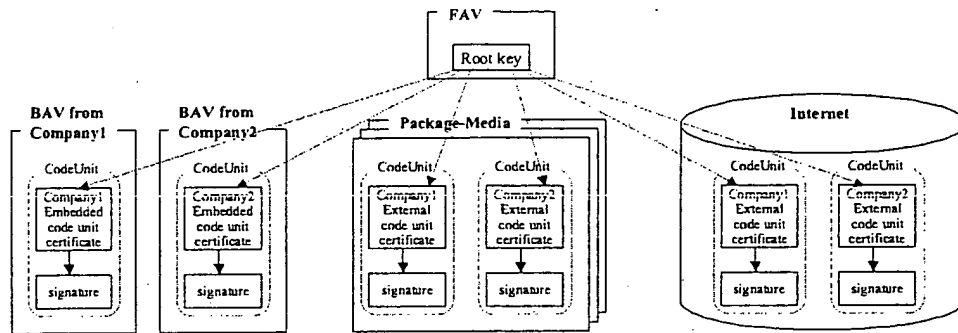
Figure 25. Certificate Tree

3.10.1.1   EMSA-PKCS1-v1_5 encoding method in HAVi

Since HAVi Authentication allows only SHA-1 hash function, the encoding method is as follows, which is compliant to EMSA-PKCS1-v1_5-Encode in PKCS#1 v2.0.

EMSA-HAVi(M, emLen)

Option: Hash   SHA-1 hash function

Input: M   Message to be encoded

        emLen   intended length in octets of the encoded message.

Output: EM   encoded message, an octet string of length emLen.

Steps:

1. Apply the hash function to the message M to produce a hash value H:

        $H = Hash(M)$2. Generate an octet string PS consisting of length $emLen - 37$ octets with value 0xFF.

3. Concatenate PS and T to form the encoded message EM as

        $EM = 01 \| PS \| 00 \| 3021300906052b0e03021a05000414 \| H$

4. Output *EM*

3.10.2 Code Unit Format

A HAVi-compliant signed code unit shall contain three specific files, named 'havi.hashfile', 'havi.signature' and 'havi.cert' in the JAR file. If the JAR file has directories, then each directory may have its own 'havi.hashfile'. Even in that case, the root directory shall have a 'havi.hashfile'.

A HAVi-compliant signed code unit may contain another specific file, named 'havi.crl', if one or more vendor-unique public key(s) have ever been revoked.

Below is the structure of HAVi-compliant signed code unit (uploadable DCM, havlet and possibly Application Module).
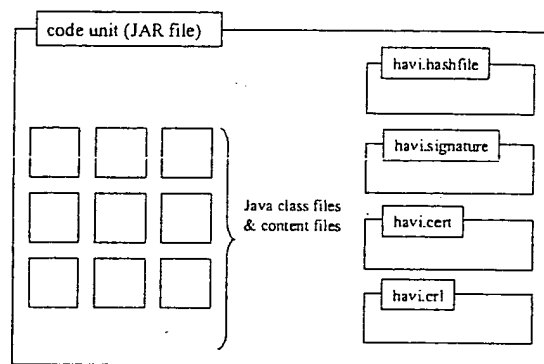
Figure 26. Authentication-specific files in a JAR file

3.10.2.1   Hash file

A code unit consists of several class files and possibly contents files. A file named 'havi.hashfile' is used to specify the filenames and their order to compute a SHA-1 Hash digest value for the directory.

The root directory shall have a master 'havi.hashfile'. If class files and/or content files are stored in a subdirectory and need to be verified, such a subdirectory also contains its own 'havi.hashfile' in it.

The format of 'havi.hashfile' is compliant to ETSI TS 101 812 V1.1.1 (2000-07), section 12.4.1.1.

However, since HAVi only allows SHA-1 as the hash algorithm, the format is shrunk as follows:

```
Syntax                                  #bit Format

Hashfile () {
    digest_count   /* always 1 */               16  uimsbf
    for( i=0 ; i<digest_count ; i++ ) {
        digest_type   /* always 2 for SHA-1 */   8  uimsbf
        name_count.                             16  uimsbf
        for( j=0 ; j<name_count ; j++ ) {
            name_length                          8  uimsbf
```

```
        for( k=0 ; k<name_length ; k++ ) {
            name_byte                               8 bslbf
        }
    }
    for( j=0 ; j<digest_length ; j++ ) {   /* always 20 for
SHA-1 */
        digest_byte                              8 bslbf
        }
    }
}
```

digest_count is set to 1 because all the files are digested by SHA-1.

digest_type is set to 2 to represent SHA-1.

digest_length is 20 (bytes) because SHA-1 generates 160 bits digest.

name_count: This value identifies the number of object names associated with the digest value. It may be one or more.

name_length: This value identifies the number of bytes in the object name

name_byte: This value holds one byte of the object name. Terminating null characters are not considered to be part of the file name.

The digest value is computed over the objects named in the havi.hashfile in the ordered list. The ordered list may contain an arbitrary mix of different object types (that is a mixture of file and directory names). The digest value is computed over the concatenated relevant data in the order. The relevant data for each objects depends on its type:

▦   if the object is a file, relevant data is the entire content of the file.

▦   if the object is a directory, relevant data is the content of the havi.hashfile of the named directory.

3.10.2.2   Signature File

The SHA-1 Hash digest value in the 'havi.hashfile' (a sequence of digest_byte) for the root-directory is signed and contained as a file named 'havi.signature'.

This file is generated by the vendor who creates the code unit, with one of the 'vendor-unique private key' issued by HAVi to the vendor.

The file format of 'havi.signature' is as follows:

MSB   LSB

offset 0
| SHA-1 Message Digest according to havi.hashfile |
| (generated by the vendor of code unit) |
| (160 bits) |

20
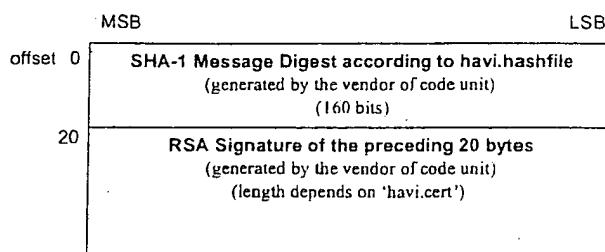| RSA Signature of the preceding 20 bytes |
| (generated by the vendor of code unit) |
| (length depends on 'havi.cert') |

Figure 27. havi.signature format

3.10.2.3   Certificate File

The signature file 'havi.signature' is verified with the 'vendor-unique public key' which corresponds to the vendor-unique private key that is used to generate the signature.

The 'vendor-unique public key' must be certified by the HCA. A file named 'havi.cert' is used for this purpose. The verifier module in an FAV can find the 'vendor-unique public key' to verify the 'havi.signature' in this file, and shall verify whether the public key is trusted.

'havi.cert' and corresponding vendor-unique private key will be issued to vendors from the HCA.

The file format of 'havi.cert' is as follows.

MSB   LSB

offset 0
| HAVi Authentication Version (8bits) |

1
| Key Attributes (8bits) |

2
| Serial No. of this Vendor Unique Public Key |
| (32bits) |

6
| Validity year 20XX (8bits) |

7
| Validity month YY (8bits) |

8
| Length (in bits) of Public Exponent (8bits) |

9
| Public Exponent |
| (1 to 8 bytes, depending on the previous field) |

| Length (in words of 32bits) of Public Modulus (8bits) |

| Public Modulus |
| (128, 192 or 256 bytes, depending on the previous field) |

| Length (in bytes) of Extended Fields (8bits) |

| Extended fields |

| RSA Signature of all the preceding bytes |
| (generated by HAVi for this vendor-unique public key) |
| (2048 bit) |

Key Attributes

7	6	5	4	3	2	1	0
R	R	R	R	R	V	EM	EX

Reserved

External code unit
Embedded code unit
Validity

Figure 28. havi.cert format

HAVi Authentication Version – the version number of HAVi Authentication Algorithm. This field shall be set to 0x01 for HAVi Specification Version 1.1.

Key Attributes – each bit indicates the characteristic of this vendor-unique key.

Serial No. of this vendor-unique public key – a serial number uniquely assigned by HAVi for the key.

Validity year – year until when this key is valid. The last 2 digits of year (i.e. 00 – 99) is used and represented as a signed byte (i.e. ranging from 0x00 to 0x63).

Validity month – month until when this key is valid. 01 – 12 is used and represented as a signed byte (i.e. ranging from 0x01 to 0x0c).

For the Key Attributes field, each bit represents as follows:.

Reserved bits – not used in this version. These bits shall be set to 0.

Validity bit – 1 if the key has validity, 0 for otherwise. If this bit is set to 0, the verifier shall neglect the following Validity fields.

Embedded code unit bit – 1 if the key may be used for Embedded code units, 0 for otherwise.

External code unit bit – 1 if the key may be used for External code units, 0 for otherwise.

Length of Public Exponent – length of the following Public Exponent field, expressed in bits.

Public Exponent – a sequence of Public Exponent (in terms of PKCS#1 v2.0) of size Length of Public Exponent

Length of Public Modulus – length of the following Public Modulus field, expressed in words of 32 bits

Public Modulus – a sequence of Public Modulus (in terms of PKCS#1 v2.0) of size Length of Public Modulus

Length of Extended Fields – length of any extended fields which may be added in the future version. This field shall be 0x00 for the current version.

Extended Fields – This field may contain additional data for succeeding versions of the specification. In this version of the specification the field is absent, i.e., length of Extended Fields is zero.

RSA Signature – a sequence of 2048 bits RSA signature for all the preceding bytes, generated by the HCA for the key.

3.10.2.4   Certificate Revocation List File

A vendor-unique key may be revoked if the key is compromised. A HAVi-signed code unit may contain a list of revoked keys with HAVi's signature on it. Such a list is called Certificate

Revocation List (CRL), and has a fixed filename of 'havi.crl' in the root directory of the code unit JAR file. The CRL file will be issued by the HCA and delivered to each vendor when needed. The CRL may contain all the keys which have ever been revoked, or may contain only limited but significant keys.

Each DCM/havlet code unit vendor shall include the most recently issued 'havi.crl', if available, in the JAR file.

The file format of 'havi.crl' is as follows:
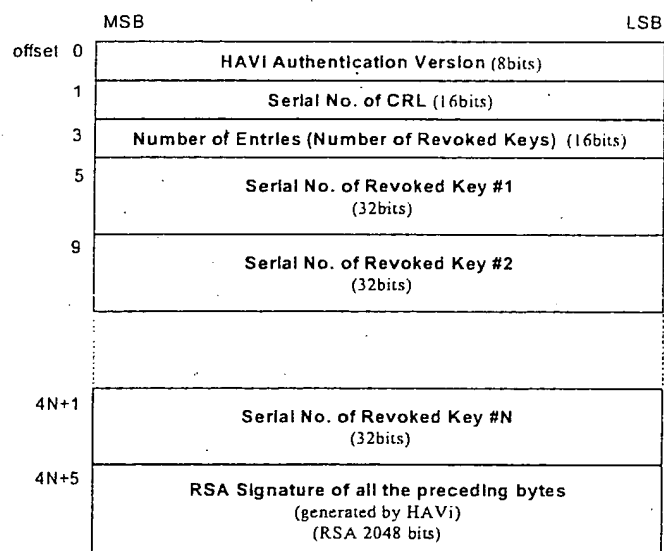
Figure 29. havi.crl format

HAVi Authentication Version – the version number of HAVi Authentication Algorithm. This field shall be set to 0x01 for HAVi Specification Version 1.1.

Serial No. of CRL – serial number of the CRL. A CRL of serial number = 0 does not exist, since 0 is reserved to mean that no keys have been revoked. When the HCA creates and delivers a new (updated) CRL, the CA will increment this value by one.

Number of Entries – indicates the number of Serial No. of Revoked Key s contained in the CRL

Serial No. of Revoked Key – serial number of each revoked key, which was assigned by the HCA and contained in corresponding 'havi.cert' file.

RSA Signature – a sequence of 2048 bits RSA signature for all the preceding bytes, generated by the HCA for the key.

3.10.2.5 Implementation note on keys, digest values and signatures encoding

Keys, signatures and digest values, when included in one of the four authentication specific files

('havi.hashfile', 'havi.signature', 'havi.cert' and 'havi.crl') shall always be seen as bit strings with leftmost bit first (bslbf format). If they need to be converted from (respectively to) integer values, primitive I2OSP (respectively OS2IP) described in PKCS#1 v2.0 shall be used. This corresponds to Big endian order.

PKCS#1 v2.0 explicitly describes all conversions that are needed regarding digest values and signatures. Key values (public exponent and public modulus) should be converted to integer values before being used in RSA algorithm.

3.10.3 Certificate Generation Procedure

When a DCM/Application Module code unit is to be signed, the necessary files are generated as follows:

* using SHA-1 Hash function, generate a 'havi.hashfile' file for the root directory of the JAR file, according to the procedure defined by ETSI TS 101 812 V1.1.1 (2000-07), section 13.4.1.1. If class files and/or content files are stored in a subdirectory and need to be verified, generation of 'havi.hashfile's for such subdirectories is needed before the 'havi.hashfile' file for the parent directory is generated.
* 'havi.cert' and corresponding vendor-unique private key will be issued to vendors by the HCA.
* using RSA algorithm, generate the digital signature for the SHA-1 Hash Digest value in the 'havi.hashfile' (a sequence of digest_byte) for the root-directory with the private key which is confidentially given to the vendor. The vendor should use a vendor-unique key of appropriate Key Attributes. Then generate a 'havi.signature' file which consists of the Hash Digest value and the signature.
* put the 'havi.hashfile' and 'havi.signature' described above, and the 'havi.cert' which corresponds to the vendor-unique key into the code unit JAR file. If one or more 'havi.crl' file(s) have ever issued by HAVi, put the latest 'havi.crl' into the JAR file.

3.10.4 Code Unit Authentication Procedure

When an FAV is to install a DCM/havlet code unit, the procedure is as follows:

3.10.4.1 DCM code unit install

* check if the code unit JAR file includes 'havi.cert', 'havi.signature' and 'havi.hashfile'. If any of these files are absent, the code unit is immediately regarded as "*untrusted*" and DCM installation fails. If all these three files are present, go to the next step.
* If the DCM manager has uploaded the code unit from the SDD of an BAV, check whether the Embedded code unit bit in the Key Attributes field is set to 1. Otherwise (the DCM Manager has downloaded the code unit from Internet), check whether the External code unit bit is set to 1. If it fails, the code unit is regarded as "*untrusted*" and DCM installation fails. If it succeeds, go to the next step.
* If Validity bit is set to 1, check whether the current date is within the validity period. If it fails, the code unit is regarded as "*untrusted*" and DCM installation fails. If it succeeds or Validity bit is set to 0, go to the next step.
* check whether the Serial No. of this Vendor Unique Public Key is identical to one of the revoked keys which the FAV maintains. If the key has been revoked, the code unit is regarded as "*untrusted*" and DCM installation fails. Otherwise, go to the next step.
* using the known (and stored in the FAV) HAVi public key, verify the 'havi.cert' (see section 3.10.4.3). If it fails, the code unit is regarded as "*untrusted*" and DCM installation fails. If it succeeds, go to the next step.
* check whether the hash values in 'havi.signature' and 'havi.hashfile' (a sequence of digest_byte) are the same. If it fails, the code unit is regarded as "*untrusted*" and DCM

installation fails. If it succeeds, go to the next step.

* using the vendor unique public key verified above, verify 'havi.signature'. If it fails, the code unit is regarded as *"untrusted"* and DCM installation fails. Otherwise, go to the next step..

* using SHA-1 Hash function, calculate a hash value according to the order specified by 'havi.hashfile'. Then compare the value with the value in the 'havi.hashfile' (a sequence of digest_byte). If it fails, the code unit is regarded as *"untrusted"* and DCM installation fails.

* if all these processes succeed, the DCM code unit is regarded as *"trusted"* and allowed to be installed.

3.10.4.2   havlet code unit install

* check if the code unit JAR file includes 'havi.cert', 'havi.signature' and 'havi.hashfile'. If any of these files are absent, the code unit is immediately regarded as *"untrusted"*. If all these three files are present, go to the next step.

* If Validity bit is set to 1, check whether the current date is within the validity period. If it fails, the code unit is regarded as *"untrusted"*. If it succeeds or Validity bit is set to 0, go to the next step.

* check whether the Serial No. of this Vendor Unique Public Key is identical to one of the revoked keys which the FAV maintains. If the key has been revoked, the code unit is regarded as *"untrusted"*. Otherwise, go to the next step.

* using the known (and stored in the FAV) HAVi public key, verify the 'havi.cert' (see section 3.10.4.3). If it fails, the code unit is regarded as *"untrusted"*. If it succeeds, go to the next step.

* using the vendor unique public key verified above, verify 'havi.signature'. If it fails, the code unit is regarded as *"untrusted"*. Otherwise, go to the next step.

* check whether the hash values in 'havi.signature' and 'havi.hashfile' (a sequence of digest_byte) are the same. If it fails, the code unit is regarded as *"untrusted"*. Otherwise, go to the next step.

* using SHA-1 Hash function, calculate a hash value according to the order specified by 'havi.hashfile'. Then compare the value with the value in the 'havi.hashfile' (a sequence of digest_byte). If it fails, the code unit is regarded as *"untrusted"*.

* if all these processes succeed, the havlet code unit is regarded as *"trusted"*.

For havlet code units, the FAV may load the "untrusted" code units into the Java runtime. However, even when loading such untrusted havlet code units is allowed, it is recommended that the FAV have a proprietary mechanism to assure that such an installation is only done with the user's responsibility.

3.10.4.3   Verifier Implementation Note

In a future version of HAVi specification, the authentication mechanism may be enhanced. In such a case, the later version will be updated so that it has backward-compatibility to this specification.

Therefore, a verifier implementation shall take it into account and comply the following rules:

* Even if the value of HAVi Authentication Version in 'havi.cert' or 'havi.crl' file is greater than 0x01, the verifier shall not regard it as a failure.

* If the value of HAVi Authentication Version in 'havi.cert' file is greater than 0x01, the value of Length of Extended Fields may not equal to 0x00. In such a case, some extended fields may be added before RSA Signature field. The verifier shall also verify the value of Length of Extended Fields and succeeding unknown sequence of bytes in verifying the signature of 'havi.cert' file. Of course the sequence of bytes does not make sense and thus will be neglected in the authentication process of this version.

3.10.5 Revocation

Each vendor is strictly responsible for keeping its own vendor unique private key confidential. In case a vendor unique private key has leaked out or compromised, some FAVs may refuse to regard a code unit which is signed by such private key as "trusted". This is so-called "revocation" process. Revocation can be performed by detecting a vendor unique public key in a 'havi.cert' which corresponds to one of such revoked vendor unique private keys maintained in the FAV during authentication procedure. HAVi defines a standard Certificate Revocation List format (See section 3.10.2.4). Each FAV shall implement a Key Revocation Mechanism.

The procedure of the key revocation mechanism for DCM and havlet code units is as follows:

* an FAV maintains whole or a part of the latest CRL with Serial No. of CRL on a persistent storage inside the FAV. An FAV shall be capable of maintaining at least 100 Serial No. of Revoked Key entries.
* the FAV will update its CRL when and only when it finds a 'havi.crl' with greater Serial No. of CRL and valid CA signature on it.
* the FAV will neglect a 'havi.crl' with lower Serial No. of CRL than the one it keeps even if it has a valid CA signature.

The loading of an HAVi-signed code unit signed with a revoked key will fail in the authentication procedure. The HAVi-signed code unit will be seen as untrusted. Revocation of keys may lead to malfunctions. Overcoming that problem is easy in the case of External code unit (a new signature with a new key is to be computed) but rather difficult in case of Embedded code units. It is thus strongly recommended for the manufacturers to have different keys to sign External code units and Embedded code units.

Loaded HAVi-signed code unit signed with a revoked key are not required to be removed by the FAV. They may thus keep to be viewed as trusted.

It might also happen the HAVi CA key to get compromised. In that case, a new HAVi key may be issued. Each company may have to get a new certificate for its current key and to change the file 'havi.cert' in all their available files. The old HAVi key shall be used neither for signing new code units nor to be embedded in new devices.

Each FAV shall have a mechanism allowing the user to turn off the security. When security will be turned off, all new downloaded code units will be seen as trusted. A FAV with an old security HAVi key will then still be able to run already loaded code units and to download new code units.

FAV may optionally have some proprietary and not publicly exposed mechanisms allowing to get the new HAVi Root key. These mechanisms shall strongly guarantee the origin of the new key. A FAV with a new Root key will behave as follows:

* All code units authenticated by the new HAVi Root key will be installed following the rules of section 3.4

* DCM code units authenticated by the old HAVi Root key will be installed following the rules of section 3.4.1 with the following restriction:

  * Embedded Code Unit bit in the Key Attributes shall be set to 1 and External Code Unit shall be set to 0.
  * An FAV with such a mechanism shall be able to store at least 2 old root keys.

3.10.6   HAVi certification procedures

The procedures related to the HAVi Certification Authority, e.g. obtaining keys and certificates, are described in [16].

7 HAVi Java API Description

7.1 Overview

To allow flexible and future-proof CE platforms, HAVi supports the uploading of Device Control Modules, Application Modules and havlets on HAVi FAV devices. The uploadable entities are written in Java bytecode and an FAV supports a Java virtual machine on which these entities can run. This chapter describes the definitions necessary to guarantee interoperability with respect to the uploading and execution of Java bytecode.

7.2 Profiles

HAVi FAV nodes support the uploading of different types of HAVi Java entities. Each FAV must be able to host DCMs of BAV devices, and so must be able to upload and execute DCM code units. Moreover, each FAV itself decides whether it uploads Application Modules or havlets. To guarantee that a HAVi Java entity shall be able to execute on each HAVi FAV to which it may be uploaded, two FAV profiles are defined indicating which classes and packages an FAV must support.

▨ Profile #1: Packages and classes that are (may be) used in DCM code units and Application Modules code units, and therefore must be supported by every FAV.

▨ Profile #2: Packages and classes that are (may be) used by havlets and therefore, need to be supported by each FAV that uploads and execute havlets. This set of packages and classes is an extension of the set of Profile #1.

Note that whether an FAV uploads havlets is a decision it makes on its own and is not influenced or ordered by other HAVi nodes in the network. Therefore, it is not necessary for other nodes, or for (DCM/Application Module/havlet) code units to know whether an FAV is of profile #1 or #2.

The uploadable code unit shall not include classes which are in the org.havi package name-space or any package name-space which begins with 'org.havi'. FAVs shall not execute any bytecode contained in such classes. The mechanism by which they achieve this (and hence the time when rejection of the class occurs) is implementation dependent. It is recommended that platforms consider using the checkPackageDefinition method of java.lang.SecurityManager to accomplish this.

7.2.1 Java API Referencing Rules

There are a number of situations in this specification where there are references from Java APIs which are mandatory in the HAVi specification to Java APIs which are not mandatory in the HAVi specification. These include the following :

▨ Methods where at least one of the arguments is a class or interface which is not mandatory in the HAVi specification.

▨ Methods whose return value is specified as a class or interface which is not mandatory in the HAVi specification.

▨ Methods which throw an exception which is not mandatory in the HAVi specification.

⊗ Fields whose type is a class or interface which is not mandatory in the HAVi specification.

In these situations, the HAVi specification does not require the method or field concerned to be present in implementations.

Their presence or otherwise is a technical and licensing issue for implementations. Implementations may include more than is specified here. However DCM code units, Application Module code units, and havlet code units are not compliant if they require more classes or interfaces than defined in this specification.

7.2.2 Profile #1: DCMs and Application Modules

The following packages and classes must be supported by each FAV and may be used in DCM and Application Module code units:

⊗ java.lang, as defined in the Java 1.1 Core API (reference [8])

⊗ java.util, as defined in the Java 1.1 Core API (reference [8])

⊗ org.havi.constants, as defined in Appendix A

⊗ org.havi.types, as defined in Appendix A

⊗ org.havi.system, as defined in Appendix A

⊗ org.havi.Iec61883, as defined in Appendix A

⊗ org.havi.fcm.*, as defined in Appendix A

⊗ java.io as defined in the Java 1.1 Core API (reference [8]) apart from the following classes File, FileInputStream, FileNotFoundException, FileOutputStream, FileReader, FileWriter, FilenameFilter and RandomAccessFile. These named classes are not mandatory in this specification.

⊗ java.net.URL, MalformedURLException as defined in the Java 1.1 Core API (reference [8])

The external forms used by objects implementing java.io.Serializable is not fixed by this specification or any of its referenced specifications. Hence there is no requirement for the classes java.io.ObjectInputStream and java.io.ObjectOutputStream to be inter-operable between HAVi implementations.

Support of java.net package is required for at least one implementation dependent protocol for use with instances of the java.net.URL class. The methods Class.getResource() and ClassLoader.getResource() shall return instances of java.net.URL using this protocol when used to access files carried in the JAR file of an uploaded code unit.

The method URL.getContent shall work as specified in its specification even though the reference to the URLConnection is not required to be valid. Except as specified in the list below, this method shall return instances of java.io.InputStream for all content types.

⊗ For images, java.awt.image.ImageProducer shall be returned

Moreover, to upload and install code units (see next section) an FAV may need to support (parts of) the following packages:

* java.util.zip, as defined in the Java 1.1 Core API (reference [8])

An FAV only needs to provide parts of these packages to properly implement the uploading and installation of HAVi code units, it does not need to provide these classes to uploaded Java entities. So, writers of DCM and Application Module code units shall not rely on the availability of the java.io and java.util.zip packages.

All interfaces added as proprietary extensions to org.havi.* shall have default (package) access. All classes, methods and fields added as proprietary extensions to org.havi.* shall have default (package) or private access.

As long as the defined contracts (specification) are respected and kept, it is an allowable option to override protected, public and package level methods in the org.havi.* package using the standard java inheritance conventions.

The following restrictions apply to the use of the java.lang package by uploaded code units. Uploaded code units shall not violate these restrictions. The behavior of FAVs should an uploaded code unit violate these restrictions is not specified. These restrictions do not remove any class or method signatures concerned from the platform.

* The following methods shall not be called:
    * Runtime.exec()
    * Runtime.load()
    * Runtime.loadLibrary()
    * Runtime.runFinalizersOnExit()
    * System.exit()
    * System.load()
    * System.loadLibrary()
    * System.runFinalizersOnExit()
    * Thread.stop()
    * Thread.suspend()
    * Thread.resume()

* The following fields shall not be used:
    * System.in

* Methods in the following classes shall not be called:
    * java.lang.Process

* Uploaded code units shall be able to use:
    * System.out
    * System.err
    * Runtime.traceInstructions()
    * Runtime.traceMethodCalls()

for debugging without any adverse effects to the code unit. The output shall not be visible to normal end users and shall not conflict with any other API or feature of the platform. It is an allowed implementation option to not generate any output. It is not an allowed implementation to stall or block until some implementation specific debugging device is connected.

* The java.lang.Compiler class and following methods shall be taken as hints from an application to the system however there is no guarantee of what happens:

*   Runtime.gc()

*   System.gc()

▪   The System.setProperties() and System.setSecurityManager() methods will always throw an exception when called by uploaded code units.

▪   SecurityManager.checkCreateClassLoader() shall always throw a SecurityException if an application attempts to create its own subclass of java.lang.ClassLoader.

7.2.3 Profile #2: Havlets

Besides the packages defined in Profile #1, the following packages must be supported by each FAV that uploads havlets and may be used in havlet code units:

▪   A subset of Java AWT as defined in section 8.2.2

▪   org.havi.ui and org.havi.ui.event, as defined in Appendix A

7.3 Mapping HAVi IDL to Java

7.3.1 Introduction

The HAVi specification uses a subset of IDL to represent data types, structures and API's (IDL operations). The intent is to provide a normative way of representing the API's. The HAVi Messaging System will send data in big-endian order and the mapping of data types is based on CDR (Common Data Representation) from GIOP (General Inter Orb Protocol) version 1.1 as described in section 3.2.3.4.

This section explains the rules applied to derive classes that form the basis of the HAVi Java APIs. The rules are mostly derived from Object Management Group's IDL to Java mapping document. However, to keep with the requirement that the HAVi Java APIs be of a small memory foot-print and provide high performance, the rules have been slightly modified. This requirement is consistent with the requirements of consumer electronic devices.

7.3.2 const

An IDL const with constructed type is mapped to a public interface with the same name as the type of the constant with the prefix "Const". The value of the const is mapped to a public static final field in the interface with the same name as the constant. All tables with constant values in the annex of this specification are mapped to appropriate "Const" interfaces.

An example of mapping a const of a constructed type to Java is given below:

IDL	JAVA

const TypeX A = 0x0005;
const TypeX B = 0x0008;
	public interface ConstTypeX {
	   public static final TypeX A = 0x0005;
	   public static final TypeX B = 0x0008;
	}

7.3.3 Basic Types

7.3.3.1   boolean

The IDL boolean type correspond to the boolean Java type.

7.3.3.2   char, wchar and octet

The IDL type char is mapped to the Java type byte.

The IDL type wchar is mapped to the Java type char. Since HAVi states that wchar size is 2 bytes long and is in the UNICODE char set then the mapping is natural. HAVi recommends using the IDL wchar type for printable characters.

The IDL type octet is mapped to the Java type byte.

7.3.3.3   string and wstring

OMG IDL defines the string type string to be consisting of all possible 8-bit quantities except null. A string is similar to a sequence of char. Hence the mapping to Java is a sequence of bytes.

The IDL type wstring is mapped to the Java.lang.String. A wstring in HAVi only contains UNICODE characters (two bytes per character)

7.3.3.4   Integers

The IDL integer types are mapped to the Java integer types as the followings:

IDL	Java
short	short
unsigned short	short
long	int
unsigned long	int
long long	long
unsigned long long	long

As the unsigned qualifier does not exist in Java, the programmer will have to take care of comparisons, which will involve an IDL integer.

The corresponding Java class (Integer, Long) will be used as help for unsigned comparisons.

7.3.3.5 *Floating Points*

The IDL floating-point types (float and double) are mapped with the corresponding Java floating point types.

7.3.4 Constructed Types

The IDL struct and union types defined in HAVi are mapped to class definitions in Java. These class definitions can be found in org.havi.types. These classes extend either the HAVi defined HaviObject abstract class or HaviImmutableObject abstract class.

The HaviObject class extends the java.lang.Object class and implements both of the org.havi.Marshallable interface and the java.lang.Cloneable interface. The HaviObject class implements an equals method, a hashCode method, a marshal method, an unmarshal method and a clone method as abstract methods.

The HaviImmutableObject class extends the HaviObject class and implements an unmarshal method as a final method which always throw the HaviUnmarshallingException exception.

If the classes extend the HaviObject class, then they implement an equal method, a hashCode method, a marshal method, an unmarshal method and a clone method as concrete methods.

If the classes extend the HaviImmutableObject class, then they implement an equals method, a hashCode method, a marshal method, and a clone method as concrete methods.

Also the classes must provide a constructor for constructing an instance of the object from a HaviByteArrayInputStream. This is further explained in the Marshalling and Unmarshalling section. Classes extending HaviImmutableObject do not permit any methods that change the member values once the object is constructed. That is, there must be no accessor methods to change any value of the object.

The equals method concerns semantic equality of objects, i.e., equality of all fields. If the argument of the equals method is null, the equals method shall return false. The hashCode method shall return a hash code value for the object and shall provide the same contract as specified in the description of hashCode method in java.lang.Object. The clone method of the class which extends the HaviObject class shall provide recursive cloning of objects and return the clone object, i.e., if object X refers to an object Y, a clone of X refers to a clone of Y. The clone method of the class which extends the HaviImmutableObject class shall return the object itself for which the clone method is called. Marshalling and unmarshalling are explained in Section 7.3.7.

7.3.4.1 *enum*

The IDL enum type is mapped to a public Java interface. The Java interface name is the name of the IDL enum type with the prefix "Const". Each enum value corresponds to a public static final field. If no specific value is provided for the enum then the value assigned will start from 0 and will be incremented in units of 1 for succeeding values.

An example of mapping an enum to Java is given below:

IDL	JAVA

enum TypeX { A, B, C }	public interface ConstTypeX{
	    public static final int A =0,
	                         B=1,

		C=2;
	}	

7.3.4.2   struct

The IDL `struct` type is mapped to a final Java class. The Java class name is the name of the IDL `struct` type. Each `struct` member corresponds to a pair of an accessor method and a modifier method for an internal private field which keeps the member value. The Java class provides a constructor to initialize the struct object and additionally a null constructor (the struct fields could be updated after the object creation) if it extends from HaviObject.

In addition all classes defined for `struct` must extend from either the abstract HaviObject class or the abstract HaviImmutableObject class. They must implement the methods equals, hashCode, marshal and clone. In addition, the unmarshal method must be implemented if the classes defined for struct extends HaviObject. Also the classes must provide a constructor for constructing an instance of the object from a HaviByteArrayInputStream. This is further explained in section 7.3.7 on marshalling and unmarshalling.

An example of mapping a `struct` to Java is given below:

IDL	JAVA
| ```
struct TypeStruct
{
    AType A;
    Btype B;
    Ctype C;
};
``` | ```
public final class TypeStruct extends HaviObject {
        private AType A;
        private Btype B;
        private Ctype C;

        public final TypeStruct(){}
        public final TypeStruct(AType _A, Btype _B, Ctype _C){
            A = _A;
            B = _B;
            C = _C;
        }

        public final TypeStruct(HaviByteArrayInputStream hi)
                throws HaviUnmarshallingException {
            ...
        }

        public void marshal(HaviByteArrayOutputStream ho)
                throws HaviMarshallingException {
            ...
        }

        public void unmarshal(HaviByteArrayInputStream hi)
                throws HaviUnmarshallingException {
            ...
        }

        public boolean equals(Object o) {
            ....
        }

        public int hashCode() {
            ...
        }

        public Object clone() {
            ...
        }
}
``` |

The behavior of accessor methods getting each struct member value, modifier methods setting each struct member value, and constructors initializing instances shall be as follows:

※ If the Java class extends the HaviObject class

    ※ If the field is a class other than subclasses of the HaviImmutableObject class, a reference to the object is got/set.

    ※ If the field is a subclass of the HaviImmutableObject class, a reference to the object is got/set.

    ※ If the field is of the basic data type, the basic data is got/set by value

※ If the Java class extends the HaviImmutableObject class

    ※ If the field is a class other than subclasses of the HaviImmutableObject class, a deep copy of the object is created and got/set.

    ※ If the field is a subclass of the HaviImmutableObject class, a reference to the object is got/set.

    ※ If the field is of the basic data type, the basic data is got/set by value

7.3.4.3 *union*

The IDL union is mapped to a final Java class with the same name that has:

    ※ a default constructor (only if extending from HaviObject)

    ※ one constructor method for each branch

    ※ an accessor method for the discriminator, named getDiscriminator()

    ※ an accessor method for each branch

    ※ a modifier method for each branch (only if extending from HaviObject)

    ※ a modifier method for each branch that has more than one case label (only if extending from HaviObject)

    ※ a default modifier method if needed (only if extending from HaviObject)

The normal name conflict resolution rule is used (prepend an "_") for the discriminator if there is a name clash with the mapped union type name or any of the field names.

The branch accessor and modifier methods are overloaded and named after the branch. Accessor methods shall raise the HaviUnionException if the expected branch is not set.

One modifier method exists for each member accepting as unique parameter the member's value; thus, the discriminator is automatically set to a legal value for that union member. One accessor method exists for each union member. It will return the value of the current member. Attempt to access a member which is not the current one (regarding the discriminator value) will generate an exception. The member's constructors initialize the discriminator to a specified value corresponding to a legal value (in case of unique non-default case label, then that label is used as the implicit discriminator). The default class constructor does not initialize the union. Thus it is necessary to use a member modifier method or member constructor method before using any accessor method.

All the classes mapped from unions will have a set of constructors which can initialize the class to any branch of the union. These constructors will have a parameter called switchType to specify the branch.

In addition all classes defined for union must extend either the abstract HaviObject class or the abstract HaviImmutableObject class. They must implement the abstract methods equals, hashCode, marshal and clone. In addition, the unmarshal method must be implemented if the classes defined for union extends HaviObject.. All classes must provide a constructor for constructing an instance of the object from a HaviByteArrayInputStream. This is further explained in section 7.3.7 on marshalling

and unmarshalling.

An example of mapping a union to Java is given below:

IDL	JAVA
| ```
union TypeUnion switch
(long)
{
   case 1: AType A;
   case 2: Btype B;
   case 3:
   default:Ctype C;

};
``` | ```
public final class TypeUnion extends HaviObject {

   private int discriminator;    // the discriminator

   private AType A;
   private Btype B;
   private Ctype C;

      public int getDiscriminator ( )   throws
HaviUnionException{...}

      public TypeUnion (HaviByteArrayInputStream hi )
         throws HaviUnmarshallingException { ... }

      public TypeUnion ( int switchType, AType _A )
         throws HaviInvalidValueException { ... }
      public void setA ( AType _A )
         throws HaviInvalidValueException { ... }
      public Atype getA ( )
         throws HaviUnionException { ... }

      public TypeUnion ( int switchType, BType _B )
         throws HaviInvalidValueException { ... }
      public void setB ( Btype _B )
         throws HaviInvalidValueException { ... }
      public BType getB ( )
         throws HaviUnionException { ... }

      public TypeUnion ( int switchType )
         throws HaviInvalidValueException { ... }

      public TypeUnion ( int switchType, CType _C )
         throws HaviInvalidValueException { ... }
      public void setC( Ctype _C )
         throws HaviInvalidValueException { ... }
      public CType getC ( )
         throws HaviUnionException { ... }

      public boolean equals ( Object o ) {
         ....
      }

      public int hashCode() {
         ...
      }

      public void marshal ( HaviByteArrayOutputStream hbaos )
                throws HaviMarshallingException {
         ....
      }

      public void unmarshal(HaviByteArrayInputStream hbais)
                throws HaviUnmarshallingException {
         ...
      }

public Object clone ( ) {
         ....
      }
``` |

401

	}

The behavior of accessor methods getting each union member value, modifier methods setting each union member value, and constructors initializing instances shall be as follows:

※ If the Java class extends the HaviObject class

     ※ If the field is a class other than subclasses of the HaviImmutableObject class, a reference to the object is got/set.

     ※ If the field is a subclass of the HaviImmutableObject class, a reference to the object is got/set.

     ※ If the field is of the basic data type, the basic data is got/set by value

※ If the Java class extends the HaviImutableObject class

     ※ If the field is a class other than subclasses of the HaviImmutableObject class, a deep copy of the object is created and got/set.

     ※ If the field is a subclass of the HaviImmutableObject class, a reference to the object is got/set.

     ※ If the field is of the basic data type, the basic data is got/set by value

7.3.4.4   sequence

The IDL sequence type is mapped to a Java array of the mapped type. Bounded sequences imply a bound check during marshalling of IDL operation parameters. Holder classes (explained later) for the sequences are defined by using the same name as the type of the sequence with "SeqHolder" appended to it. However, sequence<octet> shall be mapped into HaviByteArrayOutputStream if it needs to be marshalled using CDR. And sequence<octet> shall be mapped into HaviByteArrayInputStream if it needs to be unmarshalled using CDR (See the section 7.3.7). Otherwise sequence <octet> shall be mapped into byte[].

7.3.4.5   array

The IDL array type is mapped to the Java array. Bounded arrays imply a bound check during marshalling and unmarshalling of IDL operation parameters.

7.3.4.6   typedef

Java does not have a typedef construct.

7.3.4.6.1   Simple IDL Types

Any typedef that is a type declaration for a simple type is mapped to the original (mapped type) everywhere the typedef type appears. Thus an IDL construct typedef ushort ElementId does not generate a corresponding ElementId class. Wherever ElementId appears in the IDL it is assumed to be replaced with ushort and the appropriate mapping rules are further applied.

7.3.4.6.2   Complex IDL Types

Typedefs for non-arrays and sequences are "unwound" to their original type until a simple IDL type or user-defined IDL type (of the non typedef variety) is encountered. Typedefs for arrays and sequences are converted to corresponding named classes. Thus the IDL typedef

`sequence<octet>` Bitmap is converted to a corresponding Bitmap class. An example of a Bitmap class is given below:

```java
// Java
package org.havi.types;
public final class Bitmap extends HaviObject {
    private byte[] value;

    public Bitmap() {...}

    public Bitmap(byte[] value) throws HaviInvalidValueException {
        ...
        this.value = value;
        ...
    }

    public Bitmap(HaviByteArrayInputStream hbais) throws HaviUnmarshallingException {
        ...
        this.unmarshal(hbais);
        ...
    }

    public byte[] getValue() {
        return(value.clone());
    }

    public void setValue(byte[] value) throws HaviInvalidValueException {
        this.value = value;
    }

    public Object clone() {...}
    public boolean equals(Object o) {...}
    public int hashCode() {...}
    public void marshal(HaviByteArrayOutputStream hbaos) throws HaviMarshallingException {...}
    public void unmarshal(HaviByteArrayInputStream hbais) throws HaviUnmarshallingException {...}
}
```

In addition all classes defined for `typedef` must extend either the abstract HaviObject class or the abstract HaviImmutableObject class. They must implement the abstract methods equals, hashCode, marshal and clone. In addition, the `unmarshal` method must be implemented if the classes defined for typedef extends HaviObject. All the classes must provide a constructor for constructing an instance of the object from a HaviByteArrayInputStream. This is further explained in the Marshalling and Unmarshalling section.

7.3.5 Holder Classes

Support for `out` and `inout` parameter passing modes requires the use of additional "holder" classes. These classes are available for all the basic IDL datatypes and sequences. For user defined IDL types, such as `struct` and `union`, if the types are mutable, no additional holder classes are required. For user defined IDL types, such as `struct` and `union`, if the types are immutable, the holder class name is the Java type name (with its initial letter capitalized) to which the datatype is mapped with an appended Holder. For the basic IDL datatypes, the holder class name is the Java type name (with its initial letter capitalized) to which the datatype is mapped with an appended Holder. (E.g. IntHolder for an `int`.) For `sequence` the holder class is defined by using the same name as the type of the sequence with SeqHolder appended to it.

Each holder class must extend from the HaviHolder class. Each holder class has a pair of the accessor method named getValue and the modifier method named setValue for an internal private field named "value" which holds a basic data value, an array instance or an immutable instance. Each holder class provides a constructor to initialize the value field of a holder instance, and additionally provides a default constructor (the value field can be updated after the object creation). The default

constructor sets the value field to the default value for the type of the value field as defined by the Java language: FALSE for boolean, 0 for numeric and char types and null for arrays and other classes. Each holder class must implement equals, hashCode, marshal and unmarshal methods. In addition each holder class must provide a constructor for constructing a holder instance from a HaviByteArrayInputStream. This is further explained in section 7.3.7 on marshalling and unmarshalling. The behavior of the accessor method getValue(), the modifier method setValue(), and the constructor initializing the value field shall be as follows:

▨ If the value field is a subclass of the HaviImmutableObject class, a reference to the object is got/set.

▨ If the value field is an array class, a reference to the object is got/set.

▨ If the value field is of the basic data type, the basic data is got/set by value

An example of a holder class for a basic type and a sequence type is given below:

```java
//.Java
package org.havi.types;
public final class ShortHolder extends HaviHolder {
    private short value;

    public ShortHolder() { ...}

    public ShortHolder(short value) throws HaviInvalidValueException {

        ...
        this.value = value;
        ...

    }

    public ShortHolder(HaviByteArrayInputStream hbais) throws HaviUnmarshallingException {

        ...
        this.unmarshal(hbais);
        ...

    }

    public short getValue() {
        return(value);
    }

    public void setValue(short value) throws HaviInvalidValueException {
        this.value = value;
    }

    public boolean equals(Object o) { ... }
    public int hashCode() { ... }
    public void marshal(HaviByteArrayOutputStream hbaos)throws HaviMarshallingException { ... }
    public void unmarshal(HaviByteArrayInputStream hbais) throws HaviUnmarshallingException { ... }

}


// Java
package org.havi.types;
public final class SeidSeqHolder extends HaviHolder {
    private SEID[] value;

    public SeidSeqHolder() { ...}

    public SeidSeqHolder(SEID[] value) throws HaviInvalidValueException {

        ...
        this.value = value;
        ...
```

```
    }

    public SeidSeqHolder (HaviByteArrayInputStream hbais) throws HaviUnmarshallingException {
        ...
        this.unmarshal(hbais);
        ...
    }

    public SEID[] getValue() {
        return(value.clone());
    }

    public void setValue(SEID[] value) throws HaviInvalidValueException {
        this.value = value;
    }

    public boolean equals(Object o) { ...}
    public int hashCode() { ... }
    public void marshal(HaviByteArrayOutputStream hbaos) throws HaviMarshallingException { ... }
    public void unmarshal(HaviByteArrayInputStream hbais) throws HaviUnmarshallingException { ... }
}
```

7.3.6 Exceptions

The IDL exceptions are not supported. However, the HAVi Java APIs define a HaviException corresponding to each error code and in addition define a number of other exceptions. All exceptions corresponding to HAVi error codes in HAVi Java APIs are extended from the base class HaviException. These classes are described in detail in Appendix A. Refer to section 7.3.8.3 on Error Codes to note when exceptions are raised.

7.3.6.1 Exception Throwing and Handling

The following section applies to all packages in the org.havi package namespace with the exception of org.havi.ui and sub-packages of that package. Exceptions to be thrown in HJA are divided into the following two categories: Subclasses of the HaviException class and Subclasses of the Exception class.

Exceptions which are the subclasses of the HaviException class correspond to the return error codes of HAVi API (See Section 7.3.8.3.). An exception in this category is thrown if an error code was received from another software element or if an error code should be returned to another software element. And since a client shall interpret a received unknown error code as EUNIDENTIFIED_FAILURE, HaviUnidentifiedFailureException shall be thrown if an unknown error code is received. In addition, Java clients shall catch any HaviException to avoid termination on an unknown HaviException thrown by a HaviClient of a newer FAV (See Section 5.1.7.).

The subclasses of the Exception class are for notifying of abnormalities or exceptional events among the instances of the classes defined in HJA. There are the following six exceptions in this category:

* HaviInvalidValueException
* HaviMarshallingException
* HaviUnmarshallingException
* HaviUnionException
* HaviMsgListenerExistsException
* HaviMsgListenerNotFoundException

On execution of a constructor or a modifier method in a subclass of the HaviObject class, HaviInvalidValueException is thrown if at least one argument is invalid.

405

On execution of any constructor or any method other than constructors and modifier methods in subclasses of the HaviObject class, java.lang.IllegalArgumentException (and not HaviInvalidValueException) is thrown if at least one argument is invalid.

On execution of any constructor or any method in HJA, unless the HAVi Specification or the Javadoc explicitly states that null can be used in the constructor or method, java.lang.NullPointerException (and not HaviInvalidValueException) is thrown if at least one argument is null (regardless of whether the class which provides these constructor or method extends the HaviObject class or not.).

If HaviInvalidValueException is caught on execution of a constructor or a method in subclasses of the HaviClient class, HaviServerHelper class, or HaviListener class, the constructor or method shall handle this exception in one of the following ways:

* solve the exceptional event within the method itself
* throw the exception HaviInvalidParameterException
* send the error code EINVALID_PARAMETER to an appropriate software element

On execution of a marshal method in a subclass of the HaviObject or HaviHolder class HaviMarshallingException is thrown if marshalling fails for some reason. Some possible reasons could be, use of invalid data (e.g. marshalling a null instance) or the lack of resources (e.g. marshalling extremely long array data).

On execution of an unmarshal method or an unmarshal constructor in a subclass of the HaviObject or HaviHolder class HaviUnmarshallingException is thrown if unmarshalling fails for some reason. Some possible reasons could be, use of invalid data (e.g., the message data is shorter than expected or the first value of a sequence is negative) or the lack of resources (e.g., unmarshalling extremely long array data).

If HaviMarshallingException or HaviUnmarshallingException is caught, it is impossible to determine whether the cause of this exception is the use of invalid data or an abnormality in execution environments. Thus, the constructor or method which caught HaviMarshallingException or HaviUnmarshallingException shall handle this exception in one of the following ways:

* solve the exceptional event by the method itself
* throw the exception HaviUnidentifiedFailureException
* send the error code EUNIDENTIFIED_FAILURE to an appropriate software element

On execution of an accessor method in a subclass of the HaviObject to which union is mapped, HaviUnionException is thrown if the discriminator value is different from the value corresponding to the accessor method. For example, EventId.getSystemEid() throws HaviUnionException if this method is executed when EventId is not a system event (See Section 7.3.4.3).

On execution of the addHaviListener method in the SoftwareElement class, HaviMsgListenerExistsException is thrown if the instance of subclasses of the HaviListener class specified as argument has already been attached to the instance of the SoftwareElement class.

On execution of the removeHaviListener method in the SoftwareElement class, HaviMsgListenerNotFoundException is thrown if the instance of subclasses of the HaviListener class specified as argument has not been attached to the instance of the SoftwareElement class.

7.3.6.2   States of Instance and Arguments

When an exception is thrown from a method of an instance, the state of the instance and the state of each argument of the method shall be defined by the following rules:

1) For any classes in HJA:

1-1) When any exception is thrown by any constructor or method and unless the HAVi Specification or the Javadoc explicitly describes the states of the instance or the arguments:

   * The state of the instance which threw the exception is indeterminate.
   * The state of each argument of the method is indeterminate.

2) For subclasses of the HaviObject class, subclasses of the HaviHolder class, the HaviByteArrayOutputStream class, and the HaviByteArrayInputStream class:

2-1) When a HaviUnionException is thrown by any method:

   * The state of the instance which threw the exception does not change.

2-2) When a HaviInvalidValueException is thrown by any constructor or method:

   * The state of the instance which threw the exception does not change.
   * The state of each argument of the method does not change.

2-3) When a HaviMarshallingException is thrown by any method:

   * The state of the instance which threw the exception does not change.

3) For the SoftwareElement class:

3-1) When a HaviMsgListenerExistsException, a HaviMsgListenerNotFoundException, or any exception which is of a subclass of the HaviException class is thrown by any method:

   * The state of the instance which threw the exception does not change.
   * The set of the attached listeners of the instance which threw the exception does not change.
   * Each state of the attached listeners of the instance which threw the exception does not change.

3-2) When any exception other than a HaviMsgListenerExistsException, a HaviMsgListenerNotFoundException, or an exception which is of a subclass of the HaviException class is thrown by any method:

   * The set of the attached listeners of the instance which threw the exception does not change.
   * Each state of the attached listeners of the instance which threw the exception does not change.

4) For the subclasses of the HaviClient class and the HaviServerHelper class:

4-1) When any exception which is of a subclass of the HaviException class is thrown by any message-sending method:

   * The state of the instance which threw the exception does not change.

7.3.7 Marshalling and Unmarshalling

HAVi requires that messages are sent and received following the Common Data Representation standard (see section 3.2.3.4). To facilitate sending of Java objects around the HAVi network, all data type classes used in the HAVi Java APIs, including "holder" classes, shall implement "marshalling" of the objects into output data stream and also implement "unmarshalling" of the input data. Every class for a type shall implement the marshal and unmarshal methods. In cases of classes extending from HaviImmutableObject class, the unmarshal method shall just throw HaviUnmarshallingException. All classes shall also provide a constructor, which accepts an input data stream and constructs itself by reading it. When marshalling bounded arrays and sequences, the marshal method shall raise a HaviMarshallingException if the size of the array or the sequence exceeds

the specified limit. Similarly while unmarshalling HaviUnmarshallingException shall be raised if the incoming stream has more data then the specified limit. HaviUnmarshallingException shall be also raised when a value of unmarshalled data is invalid with respect to the class.

7.3.7.1 *HaviByteArrayInputStream & HaviByteArrayOutputStream*

HAVi Java APIs provides two stream classes that provide methods for "marshalling" and "unmarshalling" messages.

HaviByteArrayInputStream class extends the java.io.ByteArrayInputStream class and provides for reading incoming data that is in Common Data Representation (CDR) format as specified in section 3.2.3.4.

HaviByteArrayOutputStream class extends the java.io.ByteArrayOutputStream class and provides for writing out data that is in CDR format.

7.3.8 HaviClient and HaviServerHelper

7.3.8.1 *Client and Server*

In the HAVi specification, several IDL interfaces are defined. Each interface defines several operations. For example, Cmm1394::GetGuidList, where Cmm1394 is the IDL interface and GetGuidList is the operation. Each IDL interface is basically mapped to a Java client class and a server helper class. Client classes and server helper classes are explained below.

The HAVi Java APIs can be used to implement client objects (software elements which would like to reach a server's software element API, as havlets, Application Modules, DCMs, FCMs, or system software elements) and server objects (DCMs or FCMs for example).

7.3.8.1.1 Client Classes

To implement client objects, the HAVi Java APIs provide "client classes".

There are two kinds of client classes. The first kind are called "remote client classes" or simply "client classes". The second kind are called "local client classes". (Some APIs have only "remote client classes" because such APIs does not distinguish remote access and local access.)

Remote client classes provide methods to access software elements of a specified type. The methods construct appropriate messages and send them to the specified software elements. In the case of asynchronous methods the result is returned to all listeners installed by the caller. In the case of synchronous methods the resulting byte stream is parsed back and returned to the caller. All remote client classes are extended from the HaviClient class.

Local client classes provide methods to access local software elements of a specified type. In the case of asynchronous methods the result is returned to all listeners installed by the caller. In the case of synchronous methods the resulting byte stream is parsed back and returned to the caller. All local client classes are extended from the corresponding remote client class.

For example, the remote client class of the Cmm1394 API is defined as follows:

```
// Java

public class Cmm1394Client extends HaviClient {
    ...
```

}

And the local client class of the `Cmm1394` API is defined as follows:

```
// Java

public class Cmm1394LocalClient extends Cmm1394Client {
    ...
}
```

In the client classes, each IDL operation is mapped to two Java methods: one with the same name (except starting in lowercase), the other one with the same name (again starting in lowercase) suffixed with `Sync`. For example:

```
//IDL
Status Cmm1394::GetGuidList(
        out sequence<GUID> activeGuidList,
        out sequence<GUID> nonactiveGuidList)
```

```
//Java

public class Cmm1394LocalClient extends Cmm1394Client {
    ...

    public final void getGuidList(IntHolder transactionId)
        throws HaviGeneralException,
            HaviMsgException { ... }

    public final void getGuidListSync(int timeout,
            GuidSeqHolder activeGuidList,
            GuidSeqHolder nonactiveGuidList)
        throws HaviGeneralException,
            HaviMsgException,
            HaviCmm1394NotReadyException { ... }

    ...
}
```

The first method, getGuidList, will send the message that corresponds to the IDL `Cmm1394::GetGuidList` API using the Messaging System's `MsgSendRequest` API. The method does not wait for, or handle, the reply message coming back. The result is returned to all listeners installed by the caller. Therefore, the signature only has the `in` and `inout` parameter of the IDL definition. Moreover, it returns the `transactionId` used by the Messaging System in sending the message; this allows the caller to match the corresponding incoming reply later on.

The second method, getGuidListSync, sends the call via the Messaging System's `MsgSendRequestSync` API. The method waits for the reply message and handles the reply by providing the result in the `out` and `inout` parameters. Therefore, the signature provides all `in`, `out` and `inout` parameters. Moreover, the signature provides a parameter to specify the timeout to be used in the `MsgSendRequestSync` API. A value of "0" for timeout would result in the default timeout of the Messaging System.

In case of a local API such as the above case, note that this message transfer is merely a logical model and whether `MsgSendRequest` or `MsgSendRequestSync` API is actually called and the corresponding message is actually sent is implementation dependent.

Client classes only contain the methods accessible by non-system components, i.e. methods that may be accessed from any uploaded entity. They do not contain methods that can only be called by a system component and which may be handled in a proprietary way.

7.3.8.1.2 *Server Helper Classes*

To help the designer of a non-system software elements (e.g., DCMs and FCMs) the HAVi Java APIs provide "server helper classes". These classes contain methods to send back responses to incoming requests and also methods to call message back APIs (see section 5.1.1). All server helper classes are extended from the HaviServerHelper class.

A response method gets as parameters a destination SEID, a return code, a transaction ID (as defined in MsgSendResponse) and all output parameters defined in the specific IDL API. The instance of the SoftwareElement class which provides the MsgSendResponse API and the transfer mode of the MsgSendResponse API are given through the constructor of the server helper class. The SoftwareElement class is described in section 7.3.9.

The following example is the server helper class for a VCR FCM. The response method corresponding to the API vcr::Play is described. The instance of the SoftwareElement class and the transfer mode are given through the constructor of the VcrServerHelper class.

```java
// Java

public class VcrServerHelper extends HaviServerHelper {

    public VcrServerHelper(SoftwareElement se,
        int transferMode) { ... }

    public final void playResp(SEID destSeid,
            Status returnCode,
            int transactionId)
        throws HaviGeneralException,
            HaviMsgException { ... }
    ...
}
```

The following example is the server helper class for an FCM. The response method corresponding to the Fcm::SubscribeNotification API is described. The two methods corresponding to the message back for the <Client>::FcmNotification (which is implemented in the client) are also described.

```java
// Java

public class FcmServerHelper extends HaviServerHelper {

    public FcmServerHelper(SoftwareElement se,
        int transferMode) { ... }

    public final void subscribeNotificationResp(SEID destSeid,
            Status returnCode,
            int transactionId,
            HaviByteArrayOutputStream currentValue,
            short notificationId)
        throws HaviGeneralException,
            HaviMsgException { ... }

    ...

    void fcmNotification(
            SEID destId,
            IntHolder transactionId,
            OperationCode opCode,
            short notificationId,
            short attributeIndicator,
            HaviByteArrayOutputStream value)
        throws HaviGeneralException,
```

```
                     HaviMsgException { ... }

        void fcmNotificationSync(
                SEID destId,
                int timeOut,
                OperationCode opCode,
                short notificationId,
                short attributeIndicator,
                HaviByteArrayOutputStream value)
            throws HaviGeneralException,
                HaviMsgException { ... }


        ...

}
```

7.3.8.2   *Parameter Passing Modes*

IDL in parameters, which implement call-by-value semantics, are mapped to normal Java actual parameters. If an operation has a return value other than SUCCESS in a HAVi message, then a corresponding HaviXXXException is raised in the code of the operation caller (usually the client-side, but the server-side in the case of a MB, or "message back", operation). Note that an exception may be thrown at either the client-side or the server-side, but never on both sides. in parameters are not modified by the procedure.

IDL out and inout parameters, which implement call-by-result and call-by-value/result semantics, cannot be mapped directly into the Java parameter passing mechanism for basic types and sequences. This mapping defines additional holder classes for all the IDL basic and sequence types, which are used to implement these parameter modes in Java. The client supplies an instance of the appropriate Java holder class that is passed (by value) for each IDL out or inout parameter. The contents of the holder instance (but not the instance itself) are modified by the invocation, and the client uses the (possibly) changed contents after the invocation returns. In the case of user-defined types such as struct and union, there are no additional holder classes needed and the client supplies an instance of the appropriate Java class itself.

Parameters defined as out or inout in the IDL description may be modified by the procedure. This concerns the object passed as parameters as well as all (sub) objects to which the parameter refers. This allows the caller to allocate and reuse the classes needed for the reply and alleviates the need for creating new objects by the method on each call.

All types of variables (in, out and inout) shall not be changed by the environment (in a separate thread) during the complete duration of the call.

```
// IDL
module Example {
    interface Modes {
        void operation (in long inArg,
                    out long outArg,
                    inout long inoutArg);
    }
}
```

```
// Java
package Example;
public interface Modes {
    void operation(int inArg,
                IntHolder outArg,
                IntHolder inoutArg);
}
```

}

In the above the actual in parameter comes in as only an ordinary value. But for the out and inout parameters, an appropriate holder, if necessary, must be constructed. A typical use case might look as follows:

```
// use Java code
// select a target object
Example.Modes target = ...;

// get the in actual value
int inArg = 57;

// prepare to receive out
IntHolder outHolder = new IntHolder();

// set up the in side of the inout
IntHolder inoutHolder = new IntHolder(131);

// make the invocation
int result = target.operation(inArg, outHolder, inoutHolder);

// use the value of the outHolder
... outHolder.setValue() ...

// use the value of the inoutHolder
... inoutHolder.getValue() ...
... inoutHolder.setValue() ...
```

Before the invocation, the input value of the inout parameter must be set in the holder instance that will be the actual parameter. The inout holder can be filled in either by constructing a new holder from a value or by assigning to the value of an existing holder of the appropriate type. After the invocation, the client uses the outHolder.setValue() to set the value of the out parameter, and the inoutHolder.getValue() to access the output value of the inout parameter. The return result of the IDL operation is available as the result of the invocation.

7.3.8.3   Error Codes

The return error codes in the HAVi specification are mapped to exceptions of similar name prefixed with HAVi, suffixed with Exception and the beginning "E" removed and "_" removed. Each word is started with uppercase and the remaining letters are in lowercase. Thus the error code ENOT_READY is translated to a HaviNotReadyException class. If the error code belongs to a particular API then the API name is inserted immediately after the prefix HAVi. Thus Msg::ENOT_READY would translate to HaviMsgNotReadyException. The HAVi Java APIs define in addition several other exceptions. These exception classes can be found in the org.havi.types package.

Note that it might be the case that an exception is raised but that the out and inout parameters still have proper values.

7.3.8.4   Parameter Checking

7.3.8.4.1   Parameter Checking Before Sending Messages

Before any message is sent, all parameters in the message shall be checked as follows:

- On execution of a constructor or a modifier method in a subclass of the HaviObject class or a subclass of the HaviImmutableObject class:

    - when a value which is not allowed by the HAVi Specification is attempted to be set to

any "basic data type" field of the instance, a HaviInvalidValueException shall be thrown.

      when null is attempted to be set to any "class" field of the instance, a java.lang.NullPointerException shall be thrown.

▩ On execution of a message-sending method in the SoftwareElement class, a subclass of the HaviClient class, or a subclass of the HaviServerHelper class:

      ◦ when a value which is not allowed by the HAVi Specification is specified as any "basic data type" "in" parameter of the method, a HaviInvalidParameterException shall be thrown.

      ◦ when a holder instance which is specified as any "basic data type holder class", "out/inout" parameter of the method turns out to hold a value which is not allowed by the HAVi Specification, a HaviInvalidParameterException shall be thrown.

      ◦ when a holder instance which is specified as any "basic data type sequence holder class", "out/inout" parameter of the method turns out to hold a value which is not allowed by the HAVi Specification, a HaviInvalidParameterException shall be thrown.

      ◦ when null is specified as any "class" parameter of the method, a java.lang.NullPointerException shall be thrown.

7.3.8.4.2 *Parameter Checking After Receiving Message*

After any message is received, all parameters in the message shall be checked as follows:

▩ On execution of an unmarshall constructor in any classes:

      ◦ when a value which is not allowed by the HAVi Specification is attempted to be set to any "basic data type" field of the instance, a HaviUnmarshallingException shall be thrown.

▩ On execution of a synchronous message-sending method in the SoftwareElement class, a subclass of the HaviClient class, or a subclass of the HaviServerHelper class:

      ◦ when a value which is not allowed by the HAVi Specification is set as the held value of any "basic data type holder class", "out/inout" parameter of the method, a HaviInvalidParameterException shall be thrown.

      ◦ when a value which is not allowed by the HAVi Specification is set as at least one of the held values of any "basic data type sequence holder class", "out/inout" parameter of the method, a HaviInvalidParameterException shall be thrown.

7.3.9 SoftwareElement

The SoftwareElement class implements safe access to the HAVi Messaging System for a single software element. A fresh SEID is created for a software element during construction of a SoftwareElement object. To send messages, class SoftwareElement implements the Messaging System send API primitives, without the SEID sourceSeid parameter. The SEID of the source (or sender) of a message is implicitly contained in the SoftwareElement object.

To handle incoming messages, HAVi defines the notion of a HAVi listener. All HAVi listener classes extend the HaviListener class and implement one abstract method called receiveMsg. This receiveMsg method generally filters and processes a message, received from the underlying system (the message dispatcher).

A listener is installed through the addHaviListener methods of the SoftwareElement class or via the SoftwareElement(HaviListener) constructor.. A listener can specify reception of all incoming messages for the software element, or only messages originating from a particular sender SEID.

Received messages are delivered to all HaviListener objects installed for the associated

SoftwareElement object (and optionally sender SEID).

Received messages are delivered to all HaviListener objects installed for the associated SoftwareElement object (and optionally sender SEID). The receiveMsg method of the listeners are invoked in the order that the listeners have been installed.

For a given message received by a SoftwareElement object, the haveReplied argument of the listeners' receiveMsg method shall be set to false until a listener has returned true. After a listener has returned true, all following listeners' receiveMsg shall have haveReplied set to true.

In the case that the message is a request, and all listeners have returned false, the SoftwareElement shall send a response with the error code UNKNOWN_MESSAGE. If one or more listeners return true the SoftwareElement shall not send a response.

The implementor of the listener should be aware that during the time that the receiveMsg method blocks, the related SoftwareElement may not be able to process other incoming messages.

How messages incoming to a device are dispatched to the relevant SoftwareElement objects is implementation dependent. Also, whether and how a receiving Messaging System implements the noack (Msg::ETARGET_REJECT) message/mechanism is SoftwareElement (or Messaging System) implementation dependent.

A SoftwareElement object has its own thread to call the HaviListener.receiveMsg callbacks; it does not block other SoftwareElement objects or the underlying Messaging System while performing these callbacks. Consequently, there are no restrictions on the implementations of the (application provided) HaviListener.receiveMsg method: the method does not have to be treated as an interrupt. However, one must be careful when using SoftwareElement.msgSendRequestSync in the HaviListener.receiveMsg method, since this will block the calling software element until the response to the supplied request is received. While the msgSendRequestSync is blocked no other incoming messages can be processed by the software element. This may result in a deadlock.

For example, suppose that a registry is implemented in such a way that it uses msgSendRequestSync within its HaviListener.receiveMsg to forward GetElement requests to other registries. When two applications on two different nodes (*A1* and *A2*, respectively) happen to query their registries (*R1* and *R2*, respectively) at approximately the same time, a deadlock may occur. *R1* is busy handling the request of *A1* and waits for the response to the GetElement forwarded to *R2*. During this period, *R1* cannot process incoming messages. Furthermore, *R2* is busy handling the request of *A2* and waits for the response to the GetElement forwarded to *R1*. Also, during this period, *R2* cannot process incoming messages. This is a deadlock situation which will result in timeouts of the msgSendRequestSync call in both *R1* and *R2*.

A general solution is to avoid synchronous sends in cases where it might lead to deadlock, or to add sufficient threads in the Messaging System clients. For example, to avoid the scenario described above, one additional thread per Registry would suffice.

Java programs will access the class SoftwareElement in a multi-threaded fashion. To avoid unnecessary EBUSY exceptions it is preferable that the implementation of the class SoftwareElement either supports outstanding messages as defined in chapter 3.2.1.2.8 or serializes any parallel messages.

7.4 Code Units

HAVi Java code units are entities for uploading Java bytecode by FAV nodes. Basically, the format of a Java code unit is the Java archive or "JAR" format as specified in Java 1.1 [7][8]. The class

loader of an FAV resolves class and resource names relative to the root directory of the archive. For the different type of code units, i.e., DCM code units, Application Module code units and havlet code units, there are different requirements on the actual class definitions that must be in the JAR file. The way they are handled by an FAV is basically the same:

- The FAV loads the classes in code unit JAR file such that they are available for the code unit. Classes in package <none> are located in the root directory of the archive.

- The FAV finds a specific class definition with the predefined name (the different types of code units are specified below) and makes a new instance of the specified class.

- The FAV calls the install method of the newly created object with proper parameters for that type of class. Also, the FAV provides a reference to an UninstallationListener interface (see below) by which the installed object can indicate when it has uninstalled itself.

- When the installed object indicates its installation via an UninstallationListener, the FAV releases the reference to the object to allow the removal via the Java garbage collector.

To allow indication of uninstallation, the FAV provides an UninstallationListener which has the following interface (in package org.havi.system):

```
public interface UninstallationListener{

    /*
    * To be called by the installed object to indicate
    * that it has removed itself completely (removed
    * all subscriptions, unregistered and closed its
    * message handle) and that the reference to the
    * object can be removed and the object can be
    * destroyed by the garbage collector.
    */
    public void uninstalled();
}
```

7.4.1 DCM Code Units

A DCM code unit is a code unit consisting of code to install a DCM. As for all code units, the format is a Java JAR file. Specific for DCM code units is that they must contain a concrete class named DcmCodeUnit in package <none> that implements DcmCodeUnitInterface:

```
public class DcmCodeUnit
        implements org.havi.system.DcmCodeUnitInterface {
    ...
}
```

This allows the DCM manager to find the proper class for installation of the DCM code unit.

The interface DcmCodeUnitInterface is an interface defined in org.havi.system as:

```
public interface DcmCodeUnitInterface {

  public int install(GUID  nodeId, UninstallationListener listener );

  public void uninstall();
}
```

DcmCodeUnit::install

Prototype
public int install(

```
GUID nodeId,
UninstallationListener listener);
```

Parameters

    *   nodeId the GUID referring to the device the DCM corresponds to

    *   listener reference to a listener to be called on uninstallation

Description

Installs a DCM code unit. The GUID provides means to communicate with a guest device via the CMM. The install() method shall install exactly one DCM and zero or more FCMs associated with the DCM. The uninstalled() method of the provided listener is invoked by the DCM code unit to notify its installer (the local DCM Manager) that all its software elements have unsubscribed their subscriptions, have been unregistered, have closed their messages handles and released access to the corresponding guest device. This allows removal of the object by the garbage collector and a new DCM code unit to be installed for the guest device, if appropriate.

The install method should only take care of installation and should return as soon as possible (it shall not provide a thread for execution).

Return value

    *   0: if the DCM code unit has been successfully installed.

    *   1: if the installation failed and no DCM software elements have been installed.

DcmCodeUnit::uninstall

Prototype
```
public void uninstall();
```

Description

This method makes the DCM code unit abort its activities at once. The code unit takes care that all its software elements have unsubscribed their subscriptions, have been unregistered, have closed their messages handles and have released access to the corresponding guest device. It indicates its uninstallation via the uninstalled() method of the provided listener.

7.4.2 Application Module Code Units

An Application Module code unit is a code unit consisting of code to install an Application Module. As for all code units, the format is a Java JAR file. Specific for Application Module code units is that they must contain a concrete class named AMCodeUnit in package <none> that implements AMCodeUnitInterface:

```
public class AMCodeUnit
        implements org.havi.system.AMCodeUnitInterface {
        ...
}
```

This allows an FAV to find the proper class for installation of the Application Module's code unit.

The interface AMCodeUnitInterface is an interface defined in org.havi.system as:

```
public interface AMCodeUnitInterface {

    public int install(
        TargetId            targetId,
        boolean             n1Uniqueness,
        UninstallationListener listener );
```

```
public void uninstall():
}
```

AMCodeUnit::install

Prototype
```
public int install(
    TargetId targetId,
    boolean n1Uniqueness.
    UninstallationListener listener );
```

Parameters
* targetId   Target ID of the Application Module
* n1Uniqueness   indication of whether the n1 field in targetId has been assigned so as to be persistently unique to this application
* listener   reference to a listener to be called on uninstallation

Description

Installs an Application Module code unit The install() method shall install exactly one AM. The Target ID of this Application module is the Target ID provided by the caller of this method (the host on which this Application Module is installed) in the targetId parameter. The targetID has been constructed by the host according to the description given in the Target ID definition in section 5.6.2.

The Application Module constructs its HAVi Unique ID based on values of the parameters provided by the host. The targetId and the n1Uniqueness field of the HUID of this Application Module shall be the same as the targetId and n1Uniqueness field provided by the caller, the other fields of the HUID may be decided by this Application Module itself (according to the rules for HUIDs).

The uninstalled() method of the provided listener is invoked by the Application Module code unit to notify to its installer that all its software elements have unsubscribed their subscriptions, have been unregistered and have closed their messages handles. This allows removal of the object by the garbage collector.

The install method should only take care of installation and should return as soon as possible (it shall not provide a thread for execution).

Return value
* 0: if the Application Module code unit has been successfully installed.
* 1: if the installation failed and no Application Module software elements have been installed.

AMCodeUnit::uninstall

Prototype
```
public void uninstall();
```

Description

This method makes the Application Module code unit abort its activities at once. The code unit takes care that all its software elements have unsubscribed their subscriptions, have been unregistered and have closed their messages handles. It indicates its uninstallation via the uninstalled() method of the provided listener.

417

7.4.3 Havlet Code Units

A havlet code unit is a code unit consisting of Java bytecode to install a havlet. As for all Java code units, the format is a Java JAR file. Specific for havlet code units is that they must contain a concrete class named HavletCodeUnit in package <none> that implements HavletCodeUnitInterface:

```
public class HavletCodeUnit
    implements org.havi.system.HavletCodeUnitInterface {
    ...
}
```

This allows an FAV to find the proper class for installation of the havlet code unit.

The interface HavletCodeUnitInterface is an interface defined in org.havi.system as:

```
public interface HavletCodeUnitInterface {

    public int install(
        SEID  source,
        UninstallationListener listener );

    public void uninstall();
}
```

HavletCodeUnit::install

Prototype
```
public int install(
        SEID  source,
        UninstallationListener listener );
```

Parameters
* source   SEID of the source where the code unit has been uploaded from
* listener   reference to a listener to be called on uninstallation

Description

Install the havlet code unit. source must be the SEID of the source (DCM or Application Module) from which the havlet code unit has been retrieved. This provides the havlet with a communication mechanism to its source, via standard HAVi messaging. The install() method shall install exactly one havlet. The uninstalled() method of the provided listener is invoked by the havlet code unit to notify to its installer that all its software elements have unsubscribed their subscriptions, have been unregistered and have closed their messages handles This allows removal of the object by the garbage collector.

Return value
* 0: the havlet code unit has been successfully installed.
* 1: no havlet code unit has been installed.

HavletCodeUnit::uninstall

Prototype
```
public void uninstall();
```

Description

This method makes the havlet code unit abort its activities at once. The code unit takes care that all its software elements have unsubscribed their subscriptions, have been unregistered and have

closed their messages handles. It indicates its uninstallation via the uninstalled() method of the provided listener.

7.5 Isochronous Data Processing

Virtual FCMs, introduced in section 3.5.2.5, allow the construction of HAVi applications that process isochronous data. This section describes the APIs provided by HAVi for such applications.

Implementation of an FCM in general depends on whether the FCM is embedded (implemented in native code) or uploaded (implemented in Java bytecode). HAVi does not specify native APIs, so implementation of an embedded FCM (whether physical or virtual) is entirely platform dependent (however the embedded FCM must respond to the HAVi messages specified in section 5.7.3). In the case of uploaded FCMs, which are platform independent, HAVi must specify the Java APIs needed for their implementation. The org.havi.system package provides interfaces to the main HAVi system components: the Messaging System, Communication Media Manager, Event Manager and Registry. This package is sufficient for developing an uploaded physical FCM. Uploaded virtual FCMs, on the other hand, require additional support. The org.havi.system package provides sufficient functionality for implementing a control interface (note, this includes controlling the content interface) but does not support implementation of the content interface itself.

For example, consider a virtual FCM which sinks isochronous data. The Stream Manager can be used to establish a connection, resulting in data flowing to the FCM, but neither the Stream Manager nor other HAVi system components include APIs which expose isochronous data. In order to process isochronous data, virtual FCMs must use the org.havi.Iec61883 package.

The org.havi.Iec61883 package supports the IEC 61883.1 protocol for transmission of isochronous data over IEEE 1394. It consists of the following classes:

> Iec61883InputStream
> Iec61883OutputStream

Iec61883InputStream

public class Iec61883InputStream
        extends java.io.InputStream

Iec61883InputStream allows virtual FCMs to consume data from a 1394 isochronous channel. The read() method provides the virtual FCM with the payload data from IEC 61883.1 CIP packets. Depacketization of CIP packets is handled by Iec61883InputStream.

Iec61883OutputStream

public class Iec61883OutputStream
        extends java.io.OutputStream

Iec61883OutputStream allows virtual FCMs to produce data on a 1394 isochronous channel. The write() method provides the payload data for IEC 61883.1 CIP packets. Packetization and transmission timing are handled by Iec61883OutputStream.

7.5.1 An Example

The following example illustrates how the above classes could be used in the implementation of a virtual FCM. This example is not complete and is merely intended to illustrate how the org.havi.Iec61883 classes relate to each other and to give a possible template for their use.

```
import org.havi.constants;
import org.havi.types;
```

```
import org.havi.system;
import org.havi.lec61883;

class ExampleVirtualFCM extends FcmServerHelper {
    // this virtual FCM has N input plugs and M output plugs
    private lec61883InputStream[] src = new lec61883InputStream[N];
    private lec61883OutputStream[] sink = lec61883OutputStream[M];

    public void IecAttachResp(SEID destSeid, Status returnCode, int transactionId,
        IecPlug pcr, InternalPlug plug)
    {
        // check for error conditions, if ok make stream
        // first find channel, assume myDcm is  DcmClient for parent DCM
        //
        myDcm.getChannelUsageSync(new IntHolder tid, pcr, new ShortHolder channel);
        if(pcr.getDir() == ConstDirection.IN)
            src[plug.getPlugNum()] = new lec61883InputStream(channel.getValue());
        else
            sink[plug.getPlugNum()] = new lec61883OutputStream(channel.getValue());
    }
    public void iecDetachResp(SEID destSeid, Status returnCode, int transactionId,
        IecPlug pcr, InternalPlug plug)
    {
        // check for error conditions, if ok destroy stream
        if(pcr.getDir() == ConstDirection.IN)
            src[plug.getPlugNum()] = null;
        else
            sink[plug.getPlugNum() = null;
    }
}
```

7.5.2 Relationship with the Stream Manager

In the above example, the "plug control register" to be processed by ExampleVirtualFcm, is passed as a parameter in iecAttachResp. Typically this parameter is not selected by the application but rather by the Stream Manager. The DCM is then informed of this selection via Dcm::Connect. A virtual FCM should be handled, by the Stream Manager, in the same manner as a physical FCM. Thus the recommended implementation for virtual FCMs is that Dcm::Connect invoke Fcm::IecAttach with the plug control register selected by the Stream Manager. Processing this request by the FCM will result in the invocation of iecAttachResp.

7.6 Example: A DCM Code Unit and DCM (Informative)

This section gives an example of the use of the HAVi Java APIs to implement a bytecode DCM and its DCM code unit. This example is provided for informative purposes and is not intended as an implementation blueprint. The example is not a full DCM implementation, but does demonstrate use of the HAVi Messaging System and Registry by a bytecode DCM.

The example consists of the following classes:

▒ MyDcm – implements the DCM APIs

▒ MyDcmListener – the HaviListener used by MyDcm

▒ DcmCore – dispatches HAVi messages for MyDcm

▒ MyDcmCodeUnit – the code unit used to install an instance of MyDcm

7.6.1 MyDcm.java

```java
// package com.someone.dcm.test;

import org.havi.types.*;
import org.havi.constants.*;
import org.havi.system.*;

public class MyDcm {
    // fields
    //
    public HUID dcmId;
    public GUID nodeId;

    public MyDcmListener listener;
    public SoftwareElement mySe;
    public SEID mySeid;

    private Cmm1394LocalClient cmm;
    private RegistryLocalClient registry;

    private int deviceClass = ConstDeviceClass.BAV;
    private String manufacturer = "Sample";
    private String userName = "SampleName";

    // constructor
    //
    public MyDcm(HUID _dcmId, GUID _nodeId) {
        try {

            dcmId = _dcmId;
            nodeId = _nodeId;
            listener = new MyDcmListener(this);
            mySe = new SoftwareElement(listener);
            mySeid = mySe.getSeid();

            cmm = new Cmm1394LocalClient(mySe);
            registry = new RegistryLocalClient(mySe);

            // use cmm to read SDD and to initialize
            // manufacturer, userName, deviceClass
            // now set in the HAVi Registry
            //
            Attribute[] at = new Attribute[10];
            HaviByteArrayOutputStream hbaos = new HaviByteArrayOutputStream();

            hbaos.reset(); hbaos.writeHaviString(manufacturer);
            at[0] = new Attribute(ConstAttributeName.ATT_DEVICE_MANUF, hbaos);

            hbaos.reset(); hbaos.writeHaviString(userName);
            at[1] = new Attribute(ConstAttributeName.ATT_USER_PREF_NAME, hbaos);

            hbaos.reset(); hbaos.writeInt(deviceClass);
            at[2] = new Attribute(ConstAttributeName.ATT_DEVICE_CLASS, hbaos);

            // ... initialization of other Registry attributes

            registry.registerElementSync(0, mySeid, at);

            // now registered so start listening
            //
            mySe.addHaviListener(listener);
        }
        catch(Exception e) {}
    }

// examples of some DCM APIs
//
```

```
        public int getDeviceClass() {
            return this.deviceClass;
        }

        public String getUserPreferredName() {
            return this.userName;
        }

        public void setUserPreferredName(String name) {
            this.userName = name;

            HaviByteArrayOutputStream data = new HaviByteArrayOutputStream();
            data.writeHaviString(userName);

            // determine SDD offset
            //
            long offset = 0x0abcd;

            try {
                // write to SDD using the CMM client API
                //
                cmm.writeSync(0, nodeId, offset, data);
            }
            catch(Exception e){}
        }
}
```

7.6.2 MyDcmListener.java

```
// package com.someone.dcm.test;

import org.havi.types.*;
import org.havi.system.*;

public class MyDcmListener extends HaviListener {
        private MyDcm myDcm;

        // constructor
        //
        public MyDcmListener(MyDcm _myDcm) throws HaviException {
            myDcm = _myDcm;
        }

        // methods
        //

        public final boolean receiveMsg(boolean haveReplied, byte protocolType,
          SEID sourceId, SEID destId, Status state, HaviByteArrayInputStreampayload) {
            if (haveReplied) {
                return false;
            }
            DcmCore core = new DcmCore(protocolType, sourceId, destId, state, payload, myDcm);
            return core.handleRequest();
        }

}
```

7.6.3 DcmCore.java

```
// package com.someone.dcm.test;

import org.havi.types.*;
import org.havi.constants.*;
import org.havi.system.*;
import java.io.*;
```

```java
public class DcmCore {
    // fields
    //
    private byte protocolType;
    public SEID sourceId;
    private SEID destId;
    private Status state;
    private HaviByteArrayInputStream payload;
    private MyDcm myDcm;


    // constructor
    //
    public DcmCore(byte _protocolType,
      SEID _sourceId, SEID _destId, Status _state,
      HaviByteArrayInputStream _payload,
      MyDcm _myDcm) {

        protocolType = _protocolType;
        sourceId = _sourceId;
        destId = _destId;
        state = _state;
        payload = _payload;
        myDcm = _myDcm;
    };

    // methods
    //
    public boolean handleRequest{

        try {
            OperationCode opCode = new OperationCode(payload);
            if (opCode.getApiCode() == ConstApiCode.DCM) {

                byte controlFlag = payload.readByte();
                int transferMode = ConstTransferMode.RELIABLE;
                if(controlFlag == 0) {
                    // this is an incoming command
                    //
                    DcmServerHelper dcms = new DcmServerHelper(myDcm.mySe, transferMode);
                    int transactionId = payload.readInt();

                    switch(opCode.getOperationId()) {
                    case ConstDcmOperationId.GET_USER_PREFERRED_NAME:
                        Status returnCode = new Status(ConstApiCode.DCM,
                            ConstGeneralErrorCode.SUCCESS);
                        dcms.getUserPreferredNameResp(sourceId, returnCode, transactionId,
                            myDcm.getUserPreferredName());
                        return true;
                    // case ...
                    }
                }
            }
        }
        catch (Exception e){}
    }
    return false;
}
```

7.6.4 DcmCodeUnit.java

```java
// import com.someone.dcm.test.*;

import org.havi.system.*;
import org.havi.types.*;
```

```
public class DcmCodeUnit implements DcmCodeUnitInterface {
    private HUID dcmId;

    public int install(GUID nodeId, UninstallationListener listener) {
        // initialize dcmId, the HUID
        MyDcm mdcm = new MyDcm(dcmId, nodeId);
        return 1;
    }

    public void uninstall() {}
}
```

8 HAVi Level 2 User Interface

This chapter provides a specification of the Home Audio/Video Interoperability Architecture User-Interface, (also called the HAVi User-Interface). This HAVi User-Interface is designed as a "TV-friendly" user-interface framework and is explicitly designed to be suitable for use and implementation on a variety of consumer electronic (CE) devices. The application programming interfaces (APIs) for the HAVi User-Interface are contained in the org.havi.ui and org.havi.ui.event packages described in Appendix A: HAVi Java APIs. In case of conflicts between the specification and the Java APIs, the Java APIs shall be the normative reference.

8.1 HAVi User-Interface Design (informative)

The HAVi User-Interface allows applications, written in Java, to determine the user interface capabilities of its host display device, accept input from the user, draw to the screen and play audio clips. It uses a subset of the AWT as defined in the Java 1.1 Core API (reference [8]) and extends this with packages and classes specific to the HAVi platform. This subset is supported in PersonalJava as defined in PersonalJava 1.1 specification (reference [9]).

8.1.1 Remote Control

The user input model from java.awt is extended to support an optional remote control. A large number of events are optional, allowing manufacturers to customize and add value to their products.

8.1.2 Television Specific Support

HAVi also adds classes to support graphics and video display functions that are available in typical television-based systems, including: support for non-square pixels, and graphics / video overlays.

8.2 java.awt Subset

Only a subset of the Java 1.1 java.awt package is required to be present on a HAVi platform. This subset is described within this section in more detail.

8.2.1 Required Elements from AWT

Since most of the widget set in the java.awt package is not "TV friendly", these classes are not required to be present in systems supporting the HAVi UI framework. TV friendly equivalents of these are provided through the HAVi User-Interface framework, which can be extended to support alternative look and feel. Classes of the java.awt package not included in this specification cannot be expected to be present in devices supporting the HAVi User Interface framework. Interoperable HAVi applications shall not make use of these classes. Where an application uses classes which fall outside of the scope of the HAVi specification, the behavior is not determined by this HAVi User Interface specification, rather it shall be determined by the implementation of the underlying platform. This specification does not prevent a manufacturer implementing a particular device using all of AWT, and any applications intended to execute solely in a particular device may exploit any classes or packages known to be in that device, but both the device and application shall not be regarded as interoperable and shall be considered to be proprietary in nature.

The specified set of classes have been chosen such that HAVi applications can implement any missing widget functionality using these classes.

* The main base classes, such as java.awt.Component, are required in order to build the HAVi widgets.

* Other classes, such as java.awt.Color and java.awt.Font, are required for all general drawing and painting.

* The layout classes, such as java.awt.FlowLayout and java.awt.BorderLayout are retained to provide flexible layout of components on various output devices.

The classes from java.awt that are listed in Table 15 are the classes that an HAVi application author can reliably interact with, and use within a HAVi compliant application. Interoperable applications must not use any references from classes in this list to classes not in this list.

Table 15. java.awt Classes Available to Interoperable HAVi Applications

java.awt	java.awt.event	java.awt.image
Adjustable(intf)	ActionListener(intf)	ImageConsumer(intf)
ItemSelectable(intf)	AdjustmentListener(intf)	ImageObserver(intf)
LayoutManager(intf)	ComponentListener(intf)	ImageProducer(intf)
LayoutManager2(intf)	ContainerListener(intf)	ColorModel
MenuContainer (intf)	FocusListener(intf)	DirectColorModel
AWTError	ItemListener(intf)	IndexColorModel
AWTEvent	KeyListener(intf)	MemoryImageSource
AWTEventMulticaster	MouseListener(intf)	PixelGrabber
AWTException	MouseMotionListener(intf)	
BorderLayout	TextListener(intf)	
CardLayout	WindowListener(intf)	
Color	ActionEvent	
Component	AdjustmentEvent	
Container	ComponentAdapter	
Cursor	ContainerEvent	
Dimension	FocusAdapter	
Event	FocusEvent	
EventQueue	InputEvent	
FlowLayout	ItemEvent	
Font	KeyAdapter	
FontMetrics	KeyEvent	
Graphics	MouseAdapter	
GridLayout	MouseEvent	
IllegalComponentStateException	MouseMotionAdapter	
Image	PaintEvent	
Insets	TextEvent	
MediaTracker	WindowAdapter	
Point	WindowEvent	
Polygon	ComponentEvent	
Rectangle	ContainerAdapter	
Shape		
Toolkit		

The classes in Table 15 are not necessarily sufficient to enable a full implementation of a HAVi compliant device, for example a device implementing the HAVi User-Interface could be implemented using Java 1.1, Personal Java , etc., which might require additional requirements on the implementation. The specification is intentionally silent on the mechanisms used to implement the Java environment for a HAVi implementation.

8.2.2 User Input Preference Interfaces

Personal Java 1.1 includes some interfaces which are not found in JDK 1.1 but are useful for a TV friendly user-interface API. The HAVi specification includes a number of interfaces intended to allow Java applications to adapt to mouseless environments like systems operated by remote control. These input preference interfaces allow component developers to specify how users can navigate among and interact with their components. They are only available to components which inherit from org.havi.ui.HComponent. The specification is intentionally silent on the mechanisms used to implement the Java environment for a HAVi implementation. An extra HAVi specific interface org.havi.ui.HAdjustmentInputPreferred is also included.

The org.havi.ui.HNoInputPreferred interface disallows user navigation, and hence actioning, etc.

A component that implements org.havi.ui.HNoInputPreferred indicates that the user may not navigate to this component. However, note that if a component which implements this interface is extended, so that the sub-classed component may implement another "XxxInputPreferred" interface, then in all cases, this other interface may take precedence. In contrast, the method isFocusTraversable shall always return true for components implementing the interfaces org.havi.ui.HActionInputPreferred, org.havi.ui.HAdjustmentInputPreferred, org.havi.ui.HKeyboardInputPreferred, org.havi.ui.HNavigationInputPreferred and org.havi.ui.HSelectionInputPreferred.

The org.havi.ui.HKeyboardInputPreferred interface indicates that it is intended to accept component specific keyboard input from the user. Platforms without keyboards may provide another means for generating such input when this component is edited, for example, by offering an on-screen keyboard.

The org.havi.ui.HActionInputPreferred interface indicates that it is intended to be actioned by the user.

The org.havi.ui.HAdjustmentInputPreferred interface indicates that it is intended to offer increment and decrement functionality to the user.

The org.havi.ui.HNavigationInputPreferred interface indicates that it is intended to offer focus traversal between org.havi.ui.HComponents to the user.

The org.havi.ui.HSelectionInputPreferred interface indicates that it is intended to offer selection and deselection to the user.

8.3 HAVi Extensions to AWT

8.3.1 General API Issues

In this package, passing null to a method or constructor shall generate a java.lang.NullPointerException except in the following circumstances :

- Where null is explicitly documented as being an allowed parameter
- Where the class where the method or constructor is defined inherits from java.util.EventObject or java.lang.Exception

It is an allowable option to override protected, public and package level methods in HAVi using the standard java inheritance conventions.

8.3.2 User Input

Java Applications in HAVi can accept input from a keyboard, a mouse or a remote control. The

keyboard and mouse inputs are supported by functions in the java.awt and java.awt.event packages. Remote control input is provided with classes in the org.havi.ui.event package. The org.havi.ui and org.havi.ui.event packages include classes that allow the application to determine the user-input capabilities of the platform on which the application is running.

8.3.2.1   Remote Control Support

The HAVi remote control classes are extended from the java.awt.event key event classes. All of the events that are added for the remote control are optional. The remote control keys fall into two categories: colored keys and dedicated keys. The intention of these keys is to provide the user direct access to various functions; however, the platform *may* implement a virtual (on-screen) mechanism to generate these events, but shall take care in this case not to hide the application. Note that it is an implementation option if (remote control) key events are repeated.

8.3.2.1.1   Remote Control Colored Keys

Up to six colored soft keys can be included on a remote control. These are optional, and are to be identified with a color. If implemented, these keys are to be oriented from left to right, or from top to bottom in ascending order. The application can determine how many colored keys are implemented, and what colors are to be used, so that the application can match the controls.

The following identifiers are available for colored key events: VK_COLORED_KEY_0, VK_COLORED_KEY_1, VK_COLORED_KEY_2, VK_COLORED_KEY_3, VK_COLORED_KEY_4, VK_COLORED_KEY_5.

8.3.2.1.2   Remote Control Dedicated Keys

The org.havi.ui.HRcEvent class defines a number of dedicated remote control events that can be used by applications. Although none of the events in the org.havi.ui.event.HRcEvent class are required to be implemented, events for power (VK_POWER), volume up and down (VK_VOLUME_UP and VK_VOLUME_DOWN), and channel up and down (VK_CHANNEL_UP and VK_CHANNEL_DOWN) are highly recommended.

However, whilst the dedicated remote control events are themselves device independent, the precise set of dedicated keys that is implemented is device dependent. The org.havi.ui.event.HRcCapabilities class enables an application to discover which events are implemented and how these are to be labeled to match the platform implementation.

8.3.2.2   Keyboard

HAVi supports keyboards via the java.awt.event package. The events supported on a keyboard can be determined by the org.havi.ui.event.HKeyCapabilities class.

Note that systems that do not include a physical keyboard can check each component to see if it implements org.havi.ui.HKeyboardInputPreferred. If this interface is implemented, the system may enable user input of alphanumeric key events, for example, via a "soft" on-screen keyboard.

8.3.2.3   Mouse

Mouse support is optional. The presence of a mouse can be detected with the org.havi.ui.event.HMouseCapabilities class.

Mouse functionality is provided by the java.awt.event package. HAVi applications must be written in such a way that a free roaming cursor is not required for correct operation. This does not mean that a HAVi application could not implement, e.g. a drawing program, but rather that the user should not be able to put the application into a state that cannot be exited without a mouse. (A user-friendly drawing package would also notify the user that a mouse is required to use this application properly.)

8.3.2.4　User Input Capabilities

Three classes are available to determine the capabilities of the user input for a given platform: org.havi.ui.event.HKeyCapabilities, org.havi.ui.event.HMouseCapabilities, org.havi.ui.event.HRcCapabilities. Each of these classes includes a method called getInputDeviceSupported, which returns true if the particular device is known to be available.

8.3.2.5　User Input Representation

The org.havi.ui.event.HRcCapabilities class includes a method called getRepresentation, which returns an object of type org.havi.ui.event.HEventRepresentation. This class defines an event as having a known representation as a string, color or symbol, or having no supported representation. The particular text, color, or symbol can be determined by calling getString, getColor or getSymbol respectively. This allows an application to describe a button on an input device correctly for a given platform. All available events should have a text representation from getString.

The six colored key events (VK_COLORED_KEY_0 – VK_COLORED_KEY_5), if implemented, must also be represented by a color – the getColor method returns a java.awt.Color object.

Key events may also be represented as a symbol – if the platform does not support a symbolic representation for a given event, then the application is responsible for rendering the symbol itself. Application rendering of keys without a symbolic representation, but with a commonly known representation, should follow the guidelines as defined in the Javadoc definition of the class.

8.3.3 Graphics Devices and Configurations

8.3.3.1　Background

There are some specialized requirements for running applications within a consumer electronic environment, rather than the simpler situation that occurs when an Applet is displayed within a web-browser. Most notably the screen dimensions and aspect ratios are significantly different between PCs and CE devices. In the current on-screen display (OSD) graphics model of today's set-top box units, video may be output in a number of different configurations, e.g. traditional 4:3 TV display, or 16:9 widescreen TV displays, etc. The graphics resolution and aspect ratio are often locked to the video resolution and aspect ratio. If the video aspect pixel ratio changes then the graphics pixel aspect ratio may also change. Thus, there are requirements to:

※ Determine the resolution and physical characteristics of the current display device.

※ Detect modifications to the resolution and physical characteristics of the current display device.

8.3.3.2 The HAVi Screen Reference Model

HAVi provides a model for the video output from a consumer electronics device. Instances of the class HScreen represent each independent final video output signal from a device. Each independent final video output signal is made up from the sum of graphics devices, video devices and backgrounds. These are represented by instances of the classes HGraphicsDevice, HVideoDevice and HBackgroundDevice respectively. All of these classes inherit from a common parent class - HScreenDevice.

The HAVi User-Interface specification provides limited support for applications to be displayed so that they are split across multiple concurrent display devices – the HSceneFactory class allows the HGraphicsDevice to be specified in the HSceneTemplate used to generate the HScene's for the application.

8.3.3.3 The HAVi Screen Device Discovery Classes

HAVi defines a means to allow applications to discover the range of display devices available. The model followed by HAVi is based on the model used in Java2 as described by the following three classes in the java.awt package - GraphicsDevice, GraphicsConfiguration and GraphicsConfigTemplate. In HAVi, this model is generalized to apply to video devices and to background devices.

8.3.3.3.1 Querying the Configuration of a Display Device

For each display device class (HVideoDevice. HGraphicsDevice and HBackgroundDevice), there are classes whose name ends in "Configuration" which represent distinct possible configurations of a single device. Applications may obtain a list of all possible configurations of a particular device. Applications may also obtain the current configuration using the getCurrentConfiguration method. Subject to security and resource management issues, applications may also set the configuration of a device using methods found on each device class.

Applications that are interested in a particular configuration of a device can request configurations matching a specific set of constraints. The first step in this process is to construct objects whose name ends in "ConfigTemplate". Instances of these classes can then be populated with the properties by the application and then used to request a configuration supporting those properties. Properties can also have priorities attached to allow applications to express whether support for that property is required by the application, whether support for that property is only preferred by that application, whether support for that property is required to be absent or whether support for that property is preferred to be absent. In some cases, properties such as PIXEL_ASPECT_RATIO require extra information. This extra information can be provided as part of the method used to add the property to the configuration template.

The Configuration for a Device can be acquired, using the getCurrentConfiguration method. A description of this Configuration can be obtained using the getConfigTemplate method that yields a ConfigTemplate that uniquely identifies the given Configuration. Individual properties in this ConfigTemplate can then be examined using the getPreferencePriority and getPreferenceObject methods – features that are implemented will return REQUIRED, features that are not implemented will return REQUIRED_NOT. Values of some properties may also be obtained through a limited set of query methods provided on HScreenConfiguration.

8.3.3.3.2 Compatibility with Existing java.awt Methods

The java.awt.Toolkit.getScreenSize method shall be equivalent to the pixel resolution of the current configuration of the default screen device returned by HScreen.getDefaultGraphicsDevice.

The value of the java.awt.Toolkit.getScreenResolution method is implementation specific. This method shall not be used by applications.

Where the screen aspect ratio is unknown (such as in the case where a set-top box is connected to an analog display), the default aspect ratio is 4:3. In the case where an analog monitor is used with a HAVi compliant set-top box the resolution returned shall be based on the raster of the set-top box, ignoring any interpolation or other processing that may be present in the monitor.

The java.awt.Toolkit.getNativeContainer method shall return null; interoperable applications should not rely on this method.

Where an input parameter to a method call is specified to be more restrictive than its Java type allows (e.g. only a restricted set of numbers are allowed as inputs), providing values outside the allowed range shall result in a java.lang.IllegalArgumentException being thrown.

8.3.3.4   *Detecting Configuration Changes on a Display Device*

It is important for CE devices to be able to detect variations in their settings, since they may be subject to "on-the-fly" modifications of these settings, for example, they may be heavily influenced by the nature of some input video streams. Hence, the HScreenDevice class provides support for detecting when its configuration (settings) have been changed, using the HScreenDevice.addScreenConfigurationListener methods and the HScreenConfigurationListener and HScreenConfigurationEvent classes.

When an HScreenDevice's configuration is modified, then an HScreenConfigurationEvent is generated. Note that after a HScreenConfigurationEvent is obtained any HScreenConfiguration (or HScreenConfigTemplate) associated with that HScreenDevice must be reacquired to obtain the current settings for the device.

In general, a modification to the HScreenDevice might require that the displayed user-interface be modified, e.g. if the resolution has changed, or the pixel aspect ratio has been modified.

8.3.3.5   *Emulated Display Devices*

The HAVi User-Interface introduces extra sub-classes that are used to indicate that a device may perform emulations of other device capabilities:

* HEmulatedGraphicsDevice

* HEmulatedGraphicsConfiguration

Instances of these classes can be returned by the same methods that would return the corresponding class without "Emulated" in the class name. Returning the sub-class indicates that the implementation is emulating the requested configuration on one of its actual supported configurations. The class HEmulatedGraphicsConfiguration includes methods to allow applications to compare the configuration being emulated and the actual underlying configuration being really used. The extent of support for emulated configurations is a profile issue. All possible emulated configurations are not required, or guaranteed to be supported. Emulated configurations may have a significant performance penalty with respect to those supported natively on the device.

8.3.3.5.1   *Mapping from Authoring to Device Coordinates*

A special case of device emulation is the emulation of various graphics coordinate systems on a

single physical device. The HAVi User-Interface provides mechanisms that allow devices to perform such emulations, e.g. by down-sampling a high-resolution system to match the limitations of a standard definition display. Thus, authors can rely on seamless mapping between authoring and device coordinates by the use of the HEmulatedGraphicsDevice class. Authors may determine an appropriate graphics device and request the best configuration that matches their requirements, as in a standard device discovery mechanism – or examine configurations themselves (both emulation and implementation) to determine appropriate settings. The extent to which devices are required to support emulation of other coordinate systems is profile dependent.

8.3.3.6   *Integrating HAVi Video Support into Platforms*

The HAVi specification includes several classes to represent video in the user interface system. This representation of video devices only includes the display of video. The setup of the video decoder and the video pipeline is not included in this specification.

The class HVideoComponent is intended to be returned by a platform specific controller for video. In platforms based on the Java Media Framework, the Player.getVisualComponent method shall return objects of this class. The class HVideoDevice provides two hooks to platform specific APIs for this setup, the methods getVideoController and getVideoSource. On platforms based on the Java Media Framework (JMF), the getVideoController method shall return a JMF Player. The getVideoSource method shall return a platform specific class encapsulating a reference to the source of the video. Possible examples of the class to be returned here could include java.net.URL or javax.media.MediaLocator.

8.3.3.7   *Backgrounds*

The HAVi specification includes several classes to represent the background of a screen, i.e. the area that is behind the running graphics and video and not covered by those. Using the same naming convention as video and graphics, these are called HBackgroundDevice, HBackgroundConfiguration and HBackgroundConfigTemplate. The basic HBackgroundConfiguration allows applications to control a single full screen background color.

The HAVi specification includes support for more sophisticated backgrounds - still images. Applications wishing to use these shall request an HBackgroundConfiguration supporting them by using the STILL_IMAGE property in an HBackgroundConfigTemplate. If this feature is supported by the platform concerned, when such a configuration is requested, an instance of the class HStillImageBackgroundConfiguration shall be returned. This class adds two extra methods, over the standard background configuration, which support the loading of background images. This loading is done through the HBackgroundImage class.

Using the HBackgroundImage class rather than the standard java.awt.Image allows for image formats that are decoded using hardware outside of the graphics system. One specific example of this is the decoding of still MPEG I frames using an MPEG video decoder. This is a commonly used feature in some devices since it provides good quality backgrounds without using software decoders or system memory. In systems where the same underlying MPEG video decoder can be used to decode both video and MPEG I frames, this decoder shall be represented both by an HVideoDevice instance and by an HBackgroundDevice instance for each application. Where an HBackgroundDevice and an HVideoDevice both map onto the same underlying real resource in the device, the ZERO_VIDEO_IMPACT property in HBackgroundConfigTemplate shall be used to discover and limit any impact of one on the other.

    ⬚   An application requesting a HBackgroundConfiguration using a HBackgroundConfigTemplate containing the ZERO_VIDEO_IMPACT property with priority REQUIRED shall only be returned one which would have absolutely no impact on any HVideoDevice if that HBackgroundConfiguration was set for its HBackgroundDevice. For example, only systems where a separate decoder is used for HBackgroundImages from video shall return a HBackgroundImage for such a template.

    ⬚   An application requesting a HBackgroundConfiguration using a HBackgroundConfigTemplate containing the ZERO_VIDEO_IMPACT property with priority PREFERRED shall only be returned one which would have no permanent impact on any HVideoDevice if that HBackgroundConfiguration was set for its HBackgroundDevice. For example, systems where the same underlying hardware is used for decoding both video and HBackgroundImages but where once decoded, the HBackgroundImage is copied into a separate decoder memory from video and video decoding resumes, shall return an HBackgroundConfiguration in this case but not the previous case.

    ⬚ · Implementations where decoding of HBackgroundImages interrupts the video for the duration of the still image shall not support any HBackgroundConfiguration where ZERO_VIDEO_IMPACT is either REQUIRED or PREFERRED.

On implementations where the same underlying MPEG video decoder is used for both video and HBackgrounds, the most recent request of an application shall always be granted where the single underlying decoder is already being used by that application.

8.3.3.8   *Control of Screen Configurations*

The HAVi specification is silent about whether a single display device is shared between multiple applications or not. For the case where a display device may be shared between applications, it provides a mechanism for applications to assert control over the right to change the configuration of the display device. The HScreenDevice class includes methods to allow applications to reserve and release the right to control this configuration. It also allows an application to register and remove listeners for events that are generated when the applications reserve and release this right.

Applications wishing to be able to control the configuration of an HScreenDevice must define a class implementing the ResourceClient interface and pass an instance of this class to the reserveDevice method of the HScreenDevice that they wish to control. If the reserveDevice method succeeds then the application obtains control over the device configuration. When an application calls the HScreenDevice.getClient method this will return the ResourceClient passed in to the last call to the reserve method on that HScreenDevice instance.

Where there is a conflict between applications, this specification includes a mechanism to allow the platform to arbitrate between conflicting applications. The policy for this arbitration is intentionally not defined in this specification. When it is decided to remove the right to control a screen from an application, this is notified through the ResourceClient interface, the notifyRelease method will always be called. The requestRelease method will only be called when the existing owner of the resource and the application requesting the resource are authenticated to have a secure relationship of some form. This specification is silent about the details of this authentication.

8.3.4 Graphics and Video Integration

8.3.4.1 Configurations

The HAVi specification allows applications to express the relationship between video, graphics and backgrounds. The method HScreen.getCoherentScreenConfigurations allows applications to express a common set of constraints for video, graphics and backgrounds and get back a coherent answer.

In addition to this, there are several means to express constraints between video and graphics. These can be used for applications which already have running video to fit a graphics configuration to that video or which have already running graphics to fit video to that graphics. In HGraphicsConfigTemplate, the constant VIDEO_MIXING allows applications to request configurations where graphics is super-imposed above video but without any requirement for pixels to be aligned. In HScreenConfigTemplate, there are constants to allow applications to ask for configurations as follows:

- VIDEO_GRAPHICS_PIXEL_ALIGNED - video & graphics pixels are the same size and aligned

- ZERO_VIDEO_IMPACT - a new graphics configuration must not change the existing video configuration

- ZERO_GRAPHICS_IMPACT - a new video configuration must not change the existing graphics configuration

8.3.4.2 Coordinate Spaces

The HAVi specification includes a normalized screen coordinate system that represents the coordinates on the screen as floating point numbers between zero and one. This coordinate system is not pixel based. Such a non-pixel-based coordinate system enables the following:

- meaningful results, even when the graphics configuration has not been determined

- meaningful results when presented video does not have a java.awt component.

- meaningful results when the video display and the graphics display are not necessarily aligned / share the same origin / share the same resolution, etc.

This screen-based coordinate system is encapsulated in the HScreenRectangle and HScreenPoint classes. This specification is silent about conversion between normalized and video coordinates. This should be addressed as part of the API providing support for control of video.

For graphics, these conversion mechanisms are found on the HGraphicsConfiguration class, since the conversion from screen to graphics coordinates is dependent on the current graphics device settings (Configuration) – especially if e.g. the graphics resolution can be varied independently of the video resolution, etc.

The HScreenRectangle mechanisms can be used to enable the alignment of (portions of) video and (portions of) graphics. The HScreenLocationModifiedListener and HScreenLocationModifiedEvent allow mechanisms to determine if the on-screen location of an HVideoComponent is modified (rather than its relative location within its enclosing container).

8.3.4.3   Transparency between Graphics and Video

The HAVi specification includes support for applications to request transparency between graphics and video. This is provided by the getPunchThroughToBackgroundColor method on the HGraphicsConfiguration class. These methods provide a factory that enables applications to provide an opaque java.awt.Color and obtain a java.awt.Color supporting some form of transparency between graphics and video. These Color objects may be used in the drawing methods in the java.awt.Graphics class to cause video to appear in the graphics system.

8.3.5 HSceneFactory, HSceneTemplate and HScene

The HAVi User-Interface is deliberately agnostic concerning the implementation of a "coordinating" environment that provides the mechanism for a user to choose and run one or more applications. In HAVi, this "coordinating" environment is known as a "home navigation shell". Application writers cannot make any assumptions that their application GUI will always be immediately visible. For example, valid implementations of a coordinating environment might include:

* A simple "full-screen" view on a single application at any one time (with some undefined mechanism to switch between them).

* A multi-window system, where windows may obscure each other.

* A "paned" system where each application occupies an area on-screen – i.e. each application is always visible, but they may be resized if other applications are installed.

Thus, mechanisms are required to initiate an on-screen display and to indicate "user-interest", or other modifications to the rendering area. These mechanisms should:

* Enable an application to request an area on-screen – however, given the possibility of differing styles of coordinating environment, an application cannot reasonably expect that its request will always be honored perfectly, and thus, a mechanism is required to indicate preferences for the application location "on-screen".

* Indicate whether the current application is the one which the user is specifically interested in. For example, a "well-behaved" application which the user is not currently using might release, or reduce its consumption of any limited resources.

* Indicate to an application that its extent and position on-screen have been modified somehow by the home navigation shell.

* Allow an application to indicate to the system, that it requires the user's attention. For example, the system may either indicate to the user that the user should choose the indicating application, or might simply automatically switch to that application.

8.3.5.1   Requesting an Area On-screen

8.3.5.1.1   HSceneFactory and HSceneTemplate

The HSceneFactory is a factory class that is used to generate HScene objects. An application can indicate the location and dimensions of the HScene in the associated HSceneTemplate, although it is not guaranteed that the resulting HScene will necessarily match all of these preferences – since this

435

is dependent on the implementation of the controlling shell and its associated policies, etc.

The application should call HSceneFactory.resizeScene if it wishes to re-size the HScene.

8.3.5.1.2 *HScene*

An HScene is an HContainer representing the displayable area on-screen within which the application can display itself and thus interact with the user. However, HScene does not paint itself on-screen, only its added "child" components and hence there is no requirement to allocate "pixels" to the HScene directly – its only effect is to "clip" its child components. Hence, HScene may be regarded as a simple connection to the window management policy within the device, acting as a "screen resource reservation mechanism" denoting the area within which an application may wish to present a component, at some point in the future. Since an HScene is by definition not painted, i.e. it is effectively transparent, the area behind (all) HScene's in the z-ordering may be exposed by the platform as an HBackgroundDevice, and/or HVideoDevice's. However, HAVi does not require platforms to provide such device capabilities, this is platform specific. The HScene semantics for transparency need to be specified exactly on a per-platform basis, for example, on some platforms an HScene might be transparent to other HScene's due to other separate applications.

For all interoperable applications, the HScene is considered the main top-level component of the application. No parent component to an HScene should be accessible to applications. Interoperable applications should not use the getParent method in HScene, since results are implementation dependent and valid implementations may generate a run-time error.

In terms of delegation, the HScene shall behave like a java.awt.Window with a native peer implementation, in that it will not appear to delegate any functionality to any parent object. Components which do not specify default characteristics inherit default values transitively from their parent objects. Therefore, the implementation of HScene must have valid defaults defined for all characteristics, e.g. Font, foreground Color, background Color, ColorModel, Cursor and Locale.

The HScene has a null LayoutManager by default – all widgets are placed using an X, Y co-ordinate, specified by the widget.

When created an HScene is not initially visible, and a call to setVisible is required to display the HScene (and also to hide it).

The application should call HSceneFactory.dispose if it wishes to destroy the HScene (and all of its currently added Components) and therefore release their associated resources for future garbage collection by the platform.

8.3.5.2 *Modifications to the HScene: Focus and Resize events*

The HScene object accepts java.awt.event.WindowEvent's, and interprets them as a java.awt.Window, however it is not required for the home navigation shell to generate all types of java.awt.event.WindowEvent.

Applications can use the java.awt.Component.requestFocus method on the HScene to indicate to the home navigation shell that the HScene should be receiving input focus. This request should be treated as a request to make the entire application visible and ready for user input, e.g. by expanding an icon, or changing the stacking order between competing overlapping applications. The decision as to whether or whenever the HScene (application) gains the input focus is entirely platform specific in terms of policy, etc. The java.awt.Component must be visible on the screen for this request to be granted – note that visibility in this context refers to whether the application has called the HScene.setVisible method, rather than any possible non-application-defined-behavior, due

to the action of the coordinating shell hiding an application.

The java.awt.event.ComponentEvent's COMPONENT_MOVED and COMPONENT_RESIZED will be received by the HScene when the controlling shell has modified the position of the HScene or changed its dimensions on screen, respectively.

8.3.5.3   Application "user-interface" Lifecycle

⊗ Outside the scope of the HAVi User-Interface:

  * The application is acquired by the platform.
  * The application is validated and security checked (possibly including authentication, byte-code verification, etc.).
  * The virtual machine is initialized, ClassLoader created, etc.
  * The application is executed
  * If the application does not require a user-interface, then it may continue as per normal.

⊗ If a user-interface, and hence some screen resource is required, then the application traverses the HScreen, HGraphicsDevice, HGraphicsConfiguration space to determine an appropriate configuration, using HGraphicsConfigTemplate e.g. video-mixable, full-screen graphics, square pixel aspect ratio, resolution 1280 by 1024.

⊗ The application configures the HGraphicsDevice appropriately, using the setGraphicsConfiguration method.

⊗ The application requests that the HSceneFactory effectively grant it access to part of the screen for that device, using HSceneTemplate, e.g. full-screen display.

⊗ The HSceneFactory returns an appropriate HScene container within which the application can display itself.

⊗ The application uses the HScene container to add all of its components to make its user-interface.

⊗ The application may take advantage of java.awt.WindowEvent's, to determine whether it has the user's (input) focus.

⊗ The application may take advantage of the events COMPONENT_RESIZED and COMPONENT_MOVED, to determine when its HScene extent / location has been modified and to tailor its presentation accordingly.

⊗ The application resizes the HScene by using HSceneFactory.resizeScene – if it wishes to resize the HScene, with the caveat that this may not be allowed by the external environment, e.g. due to window-manager policy, etc.

⊗ The application terminates its on-screen presentation by calling the HScene.dispose method

⊗ Outside of the scope of the HAVi User-Interface:

  * The application itself terminates.

8.3.6 Effects and Visual Composition using Component Mattes

8.3.6.1   Component Mattes

With org.havi.ui, the user interface is constructed from a set of components arranged in a hierarchy. The root of the hierarchy is an instance of HScene, leaf nodes are instances of HComponent and intermediate nodes are instances of HContainer. Components within a container are ordered from back to front. An example is shown in Figure 39, where *c3*, a container, is the back most component and *c1* the front most.
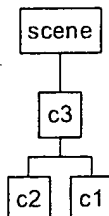
Figure 39. Scene Hierarchy

With the HMatte interface, the scene hierarchy can be modified by the inclusion of mattes (additional alpha sources), potentially for each member, i.e.:
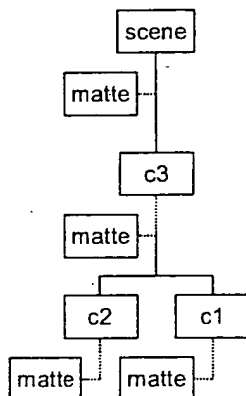
Figure 40. Scene Hierarchy with Mattes

The mattes influence the rendering of the scene, their operation can be visualized using a 2½D or layering model. The example below corresponds to the hierarchy in Figure 40 (for simplicity, the scene matte is not shown).
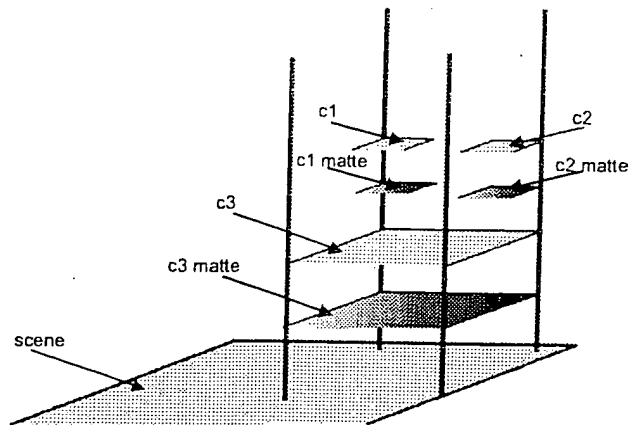
Figure 41. Component Mattes

Where pixels in a component already have an alpha value (e.g., from a PNG image), the alpha value from the component and the alpha value from the matte are multiplied together to obtain the actual alpha value to be used for that pixel.

8.3.6.2   Component Grouping

A container is either "grouped" or "ungrouped". When a container is ungrouped, its matte only influences the appearance of those regions of the container not covered by members of the container (i.e., exposed regions of the container's background). When a container is grouped, its matte influences the appearance of its background and all members of the container. For example, grouping a container and setting its matte to indicate 50% transparency will fade the container's background and all members of the container. If it is ungrouped only the background will fade.

An HContainer may be rendered as follows:

▦ If the container is ungrouped, the container's background is first rendered and then composited with the container's matte (i.e., the RGBA value of the container's background is combined with the alpha value from the matte). Then, in back to front order, each member of the container is rendered, composited with its matte, and then composited with the container.

▦ If the container is grouped, the container's background is first rendered. Then, in back to front order, each member of the container is rendered, composited with its matte, and then composited with the container. The result is then composited with the container's matte.

After an HContainer is rendered, it is composited with its parent. Compositing of an HScene is determined by the configuration of display devices.

8.3.6.3　Examples of Mattes and Component Composition
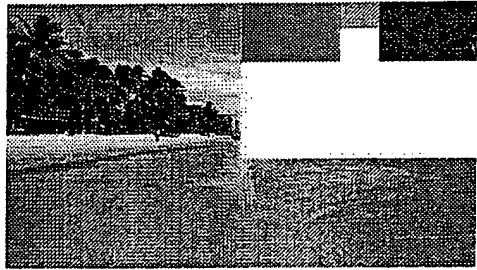
a) Bar matte for lower component.
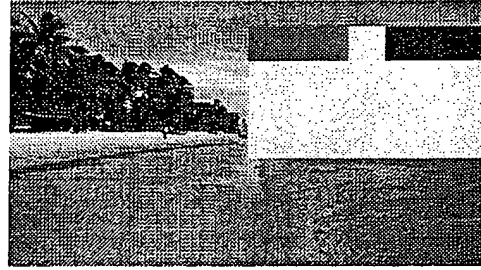
b) As in a), with top components grouped to lower.
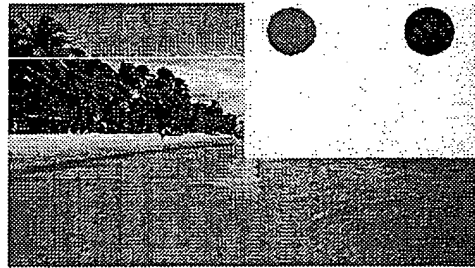
c) Circular mattes for top components.
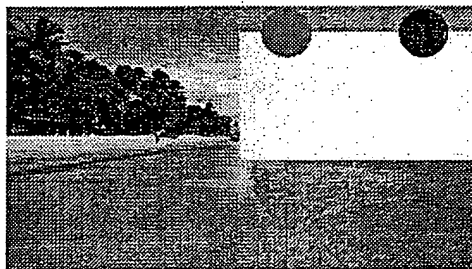
d) As in c), with bar matte for lower component.
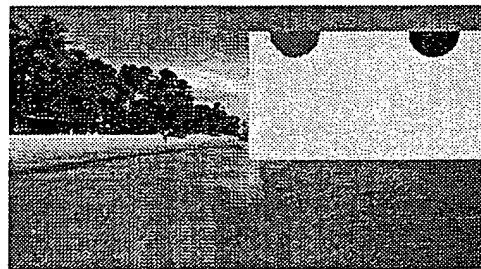
e) As in d), with top components grouped to lower.

Figure 42.　Visual Composition Examples

8.3.6.4　Effects

A great variety of effects (e.g., wipes and fades) can be performed by using *matte animations* —

sequences of mattes where the "active" element is changed over time. Matte animations can be combined with other techniques, such as component movement, to produce additional effects. The construction of matte animations is facilitated by the following classification of mattes:

HFlatMatte – the matte is constant over space and time, it can be specified by a float (0.0 is fully transparent and 1.0 fully opaque)

HImageMatte – the matte varies over space but is constant over time, it can be specified by an "image mask" (a single channel image) where the pixels indicate matte transparency

HFlatEffectMatte – the matte is constant over space but varies over time, it can be specified by a sequence of floats

HImageEffectMatte – the matte varies over space and time, it can be specified by a sequence of image masks

8.3.6.5 *Matte Sizes and Offsets*

When a HImageMatte or HImageEffectMatte is assigned to a component, the associated image (or images) is by default aligned with the component so that their origins – the pixel at (0,0) – coincide. The offset of the matte with respect to the component can be altered using the setOffset method of HImageMatte and HImageEffectMatte. Regions of the component outside the matte (resulting from either a matte being smaller than the component, or from shifting the matte) are not matted.

8.4 HAVi Widget Framework

The HAVi widget framework is designed to allow maximum flexibility to implementers of applications. It also provides the necessary extensibility to allow the widget framework to be used as the basis for other application types, such as broadcast applications. By default, the HAVi widget framework only copies object references, and does not clone objects. Cases where objects are clone'd shall be marked explicitly.

8.4.1 HAVi Event Mechanism

The HAVi event mechanism is composed of the seven classes listed in Table 16:

Table 16. HUI Events

Event	Use
HActionEvent	Interact with a component implementing the HActionInputPreferred interface
HFocusEvent	Interact with a component implementing the HNavigationInputPreferred interface
HRcEvent	Provide remote control event capabilty
HKeyEvent	Interact with a component implementing the HKeyboardInputPreferred interface
HAdjustmentEvent	Interact with a component implementing the HAdjustmentValue interface
HItemEvent	Interact with a component implementing the HSelectionInputPreferred interface
HTextEvent	Interact with a component implementing the HKeyboardInputPreferred interface

These classes serve as the mechanism by which HAVi components inform each other of event occurrences. They are not intended to be generated from applications.

441

A HAVi widget must respond to these events in addition to other applicable user-input mechanisms. However, interoperable widgets must not respond to specific key codes received through the Java AWT KeyEvent mechanism.

HXXXEvents are generated and dispatched by the HComponent base class. For example, this class must intercept suitable Java key events and generate HKeyEvents from them. This means that widgets will receive two events - the original KeyEvent and a new HKeyEvent. Although it is possible to "discover" the platform-specific implementation of HXXXEvents via this mechanism, interoperable widgets may not use this information. Widgets may ignore the KeyEvent in favor of only handling the HKeyEvent.

8.4.2 Abstraction of "Feel"

In order to provide the necessary flexibility, the HAVi User-Interface widget framework is defined around a core of abstract *Component Behaviors*. These effectively define the functionality (or "feel") of each widget which is derived from one of the Component Behaviors. Behaviors are defined for widget types having a number of states, which may be used to mimic the behavior of typical widgets.

In summary these Component Behaviors are:

* HVisible – Behavior providing basic display functionality.

* HNavigable– Behavior enabling widgets to receive navigational focus, and to define some kind of display change associated with focus change.

* HActionable– Behavior providing functionality to be invoked in response to an action.

* HSwitchable– Behavior allowing a widget to be actioned and to retain internal state information in addition to simple action behavior.

* HAdjustmentValue, HItemValue, HTextValue - Behaviors permitting the definition of widgets that return values to applications in response to user interaction.

Based upon these fundamental abstract Behaviors, all necessary HAVi functionality can be provided through derived concrete widgets, either for the provision of HAVi specific user-, interfaces, or for HAVi specific widgets. In addition, these abstractions form the basis upon which other interactive applications may be built without the requirement for the use of HAVi specific widgets. Thus, the HAVi widget framework is more generally applicable to interactive application execution, rather than exclusively focused upon HAVi. The Javadoc describes these states in more detail, including any valid DISABLED states.

8.4.3 Framework Class Hierarchy

The HAVi widget framework consists of a base class (HVisible) and a set of interfaces that model the behaviors different types of widget may exhibit. The behavior is modeled on the number of states a widget may represent. For each such state a widget can present a particular representation (graphical, textual and sound) to the user.

The widget framework allows for simple user interface development by application authors. It also reduces the size of the developed application, since most of the presentation and interaction

capability is resident on the device – developers can concentrate on the specific functionality of their application.

8.4.3.1   HContainer

Components in the HAVi User-Interface are explicitly allowed to overlap each other. Hence, the HAVi User-Interface extensions adds additional Z-ordering related methods to org.havi.ui.HContainer:

Additional semantics related to transparency of the HContainer itself and its Components, are also defined via the HMatteLayer interface.

The org.havi.ui.HContainer class also adds the ability to determine whether hardware double buffering is present, using the isDoubleBuffered method.

The org.havi.ui.HContainer class also adds the ability to determine whether it is completely opaque, by applications overriding the isOpaque method.

Additionally, the default LayoutManager for HContainer is defined to be null, i.e. absolute positioning, in contrast to the FlowLayout used in java.awt.Container.

8.4.3.2   HComponent

The base class for all HAVi widgets.

The org.havi.ui.HComponent class extends java.awt.Component to include additional semantics related to transparency of the HComponent, defined via the HMatteLayer interface.

The org.havi.ui.HComponent class also adds the ability to determine whether hardware double buffering is present, using the isDoubleBuffered method.

The org.havi.ui.HComponent class also adds the ability to determine whether it is completely opaque, by applications overriding the isOpaque method.

8.4.3.3   HVisible

Represents a widget that has only two states, for example HStaticText or HStaticIcon. This widget can be in either a "normal" state or a "disabled" state.

8.4.3.4   HNavigable

An interface that is implemented by classes that are derived from HVisible for adding an additional state that is used to indicate if the widget is currently focused.

The HNavigable interface also provides the functionality necessary to manage the focus navigation between widgets assuming a remote control style UP, DOWN, LEFT, RIGHT form of navigation, using the setFocusTraversal method.

The precise semantics of the HNavigable interface are defined in the supporting Javadoc.

8.4.3.5　HActionable

The HActionable interface extends HNavigable by adding an additional state that is used to indicate when the widget has been actioned.

The HActionable interface provides the functionality necessary to associate HActionListeners with the widget, using the addHActionListener and removeHActionListener methods. These HActionListeners will be called when the widget is actioned.

A widget that implements the HActionable interface is actioned when it receives a havi.ui.event.HActionEvent key event. The widget will move into its Actioned state by presenting its Actioned look. Any associated HActionListeners will be called by the widget calling its HActionInputPreferred.processHActionEvent method. When the HActionListeners have returned the widget will return to its focused state.

The precise semantics of the HActionable interface are defined in the supporting Javadoc.

8.4.3.6　HSwitchable

The HSwitchable interface extends HActionable by adding an additional state that is used to maintain an internal (on/off) value.

The state transitions for HSwitchable are as follows:

———▶ Focus Event Transition

············▶ Enabled/Disabled Transition

··········▶ Action Event Transition
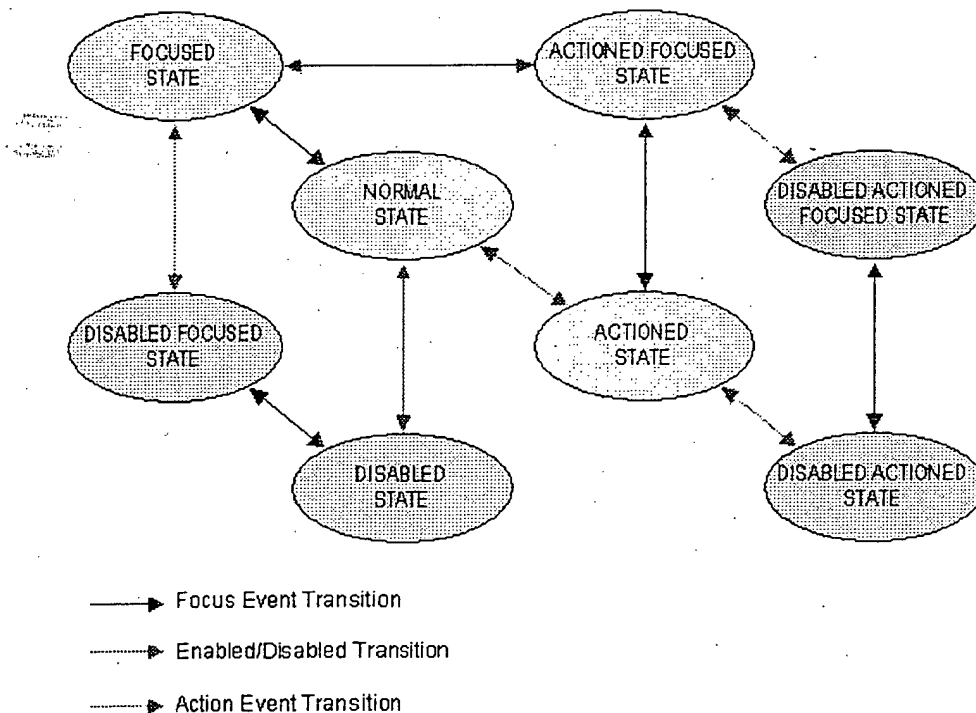
Figure 43.　HSwitchable Transitions

The precise semantics of the HSwitchable interface are defined in the supporting Javadoc. Note that any state is permitted to change to the "disabled" state.

8.4.3.7   HAdjustmentValue, HItemValue, HTextValue

These interfaces extend HNavigable by adding support for managing a widget with an internal value that can be manipulated by user interaction.

All HAVi UI components that require adjustable numerical values, such as range controls, have implemented the HAdjustmentValue interface. Here, the value responds to unit & block increments, and has an optional sound associated with such adjustments.

All HAVi UI components that have selectable content, such as list groups, have implemented the HItemValue interface. An optional sound can be associated with item selection.

All HAVi UI components that have editable text content, such as text entry controls, have implemented the HTextValue interface.

The precise semantics of the HValue interface are defined in the supporting Javadoc.

8.4.4 Separation of "Look"

The flexibility of the HAVi widget framework is further enhanced by separating the "look" component from that for "feel". This allows easy construction of many styles of presentation associated with each of the abstract Component Behaviors defined previously.

Content can be associated with each state of a widget. For each widget state, textual, graphical and user defined content can be associated with the widget. The HLook interface defines the mechanism by which the content for the particular state of the widget can be rendered.

The HLook method showLook is used to provide the rendering of the content for the widget. Note that since this method is separated from the widget class, there is no need to subclass the widget to change its look. The showLook method is responsible for repainting the entire component, including its background, subject to the clipRect of the Graphics object passed to it. The showLook method should not modify the clipRect of the Graphics object that is passed to it.

An HLook may also provide some form of border decoration, for example, drawing a rectangle around the widget when it has focus. To allow for predictable layout and presentation the HLook interface provides methods that are used to indicate the size of such a border area.

To support layout managers the HLook interface also defines the following methods, which allow the associated HVisible to query the HLook for its maximum, minimum and preferred sizes: getMaximumSize, getMinimumSize, getPreferredSize.

8.4.5 Pluggable Looks

The HAVi Widget framework provides a set of standard classes that implement the HLook interface. These can be regarded as the set of default looks that will be provided by all implementations. The particular rendering of a look is not defined and is manufacturer dependent.

Pluggable Look is defined in such a way as to allow implementers to extend the number of "looks" available to their application. By doing so, new "looks" are automatically available for every Component Behavior and for all widget types derived from those Behaviors. This is described in the Pluggable Look Interface.

To facilitate application development and to limit the size of applications, a set of pre-defined "looks" is provided. These are:

445

▓ HAnimateLook – presentation of an animated image sequence.

▓ HGraphicLook – presentation of graphical content.

▓ HRangeLook – presentation of a value within a range.

▓ HListGroupLook - presentation of both the ListGroup itself and the items held on the list.

▓ HTextLook – a simple presentation mechanism for textual content.

▓ HSinglelineEntryLook – presentation of a single line of textual content that can be edited by the user.

▓ HMultilineEntryLook – presentation of multiple lines of textual content that can be edited by the user.

This basic set allows the construction of most typical interactive user interfaces when used in conjunction with the Component Behaviors to define a widget set. It can however be extended in a general fashion to provide new categories of "look".

When a widget is constructed, it is provided with a default look. This default HLook will be one of the standard set of looks listed above. For example, the HGraphicButton is created with the HGraphicLook by default. The default look that is used when the widget is constructed can be changed by calling the static method setDefaultLook that is provided on all widget types. Any widget of that type created after the call will be created with the new HLook that was passed in as the parameter to setDefaultLook. The look of an individual widget can be modified by using the method HVisible.setLook.

The Pluggable Look mechanism is flexible enough so that the application developer can create new HLooks. For example a combined HGraphicLook and HTextLook, where the Text may overlay the Graphic, or be shown in place of the Graphic while the Graphic is being loaded.

8.4.6 Content Behavior

Content is associated with the widget through the following methods on HVisible: setTextContent, setGraphicContent, setAnimateContent and setContent. Hence, multiple content (Text, Graphics, Animations and user-defined content) can be associated with a widget. The way multiple content is rendered is dependent on the HLook associated with the widget. The default looks provided by the platform may not render all the content types.

Different content can be associated with the different states of the widget. For example, an HGraphicButton might have six different images to represent its six different states (NORMAL_STATE, FOCUSED_STATE, ACTIONED_STATE, ACTIONED_FOCUSED_STATE, DISABLED_STATE and DISABLED_FOCUSED_STATE) to the user, using the setGraphicContent. The same content can be applied to all states of the widget by using the HState constant ALL_STATES when calling setTextContent, setGraphicContent, setAnimateContent and setContent.

▓ By default, content associated with a widget is not modified to fit the dimensions of the widget. Refer to the alignment and scaling methods in HVisible to address this issue.

Mechanisms are also available that allow (graphic) content to be resized to match the widget dimensions, etc.

8.5 HAVi Resident Widgets

Using the Component Behaviors (HVisible, HNavigable, HActionable and HSwitchable) defined in the previous section a set of resident widgets is provided.

Note that implementations of the HAVi widget set shall be implemented (and behave) as lightweight components. HAVi widgets do not include an associated peer class, irrespective of the exact mechanism for their implementation, i.e., directly implemented in Java, or via some platform specific mechanism.

8.5.1 Simple Text/Graphic/Animate Widgets

The HAVi set of resident widgets includes both visible and navigable versions of the HText, HIcon and HAnimation classes:

※ Applications providing simple "display-only" non-navigable text, image, or animations may employ the "Static" versions of these classes.

※ Applications wishing to provide additional feedback, e.g. "tooltips", or audio feedback – for example, a commentary – may employ the navigable versions of these classes.

Widget Type	Description	Static	Navigable
Animation	Displays a simple sequence of images	HStaticAnimation	HAnimation
Text	Displays a text label	HStaticText	HText
Graphic	Displays an Image	HStaticIcon	HIcon

Refer to the supporting Javadoc for a more detailed description of these widgets.

8.5.2 Buttons

The HAVi set of resident widgets includes both textual and graphical version of a push button: HTextButton and HGraphicButton. These buttons implement the HActionable interface that defines their behavior.

The HToggleButton is used to represent a graphical control that has a boolean state that can be toggled on and off by the user (e.g. Checkbox or Radio Button). The HToggleButton implements the HSwitchable interface that defines its behavior. A HToggleButton widget does not have an associated text label as part of the widget. If a text label is required, a separate HStaticText widget should be created.

A set of HToggleButtons can be associated with a HToggleGroup. A HToggleGroup will ensure that a maximum of one HToggleButton is chosen at any time (i.e. a group of radio buttons).

Refer to the supporting Javadoc for a more detailed description of these widgets.

8.5.3 Range Widgets

The HAVi set of resident widgets include a group of controls to represent a particular integer value in a range of values i.e. a slider control, or scroll bar. The HStaticRange widget is a non navigable widget, HRange widget is navigable (implements the HNavigable interface) and the HRangeValue is

navigable and its value can be modified by user interaction (implements the HAdjustmentValue interface).

Refer to the supporting Javadoc for a more detailed description of these widgets.

8.5.4 List Widgets

A HListGroup is a visible that manages a dynamic set of vertically or horizontally scrollable HListElements, allowing either single or multiple HListElements to be chosen by the user. The HListGroup will automatically scroll the HListElements when the user navigates to an element that is currently not visible within the list group.

Refer to the supporting Javadoc for a more detailed description of these widgets.

8.5.5 Text Entry Widgets

The HSinglelineEntry component allows a user to enter a single line text string. A typical rendering is as a text entry field, e.g. with an associated on-screen keyboard. The HMultilineEntry widget extends the HSinglelineEntry widget and allows text to be entered over multiple lines. Both these widgets implement the HTextValue interface resulting in the widgets firing HTextEvents when starting to edit, finishing editing, and whenever the content changes.

Refer to the supporting Javadoc for a more detailed description of these widgets.

8.6 Profiles

Implementations of the HAVi User-Interface on an FAV should:

▨ have a minimum screen resolution of 320 by 240 pixels (quarter VGA)

▨ include support for the image and sound content types, as defined in DDI.

▨ either provide a physical keyboard or provide a virtual keyboard supporting at least the entry of alphanumeric codes

8.7 General Approach to Error Behavior

Where a method call is specified as taking an object as one of its input parameters, if null is passed as a parameter and not explicitly identified as a valid input for that parameter of that method then a java.lang.NullPointerException shall be thrown. Where an input parameter to a method call is specified to be more restrictive than its Java type allows (e.g. only a restricted set of numbers are allowed as inputs), providing values outside the allowed range shall result in a java.lang.IllegalArgumentException being thrown.

8.8 Register of Constants

```
public final static int org.havi.ui.HAdjustableLook.ADJUST_THUMB = -6;
public final static int org.havi.ui.HAdjustableLook.ADJUST_PAGE_MORE = -5;
public final static int org.havi.ui.HAdjustableLook.ADJUST_PAGE_LESS = -4;
public final static int org.havi.ui.HAdjustableLook.ADJUST_BUTTON_MORE = -3;
public final static int org.havi.ui.HAdjustableLook.ADJUST_BUTTON_LESS = -2;
public final static int org.havi.ui.HAdjustableLook.ADJUST_NONE = -1;
```

```
public final static int org.havi.ui.HAnimateEffect.REPEAT_INFINITE = -1;
public final static int org.havi.ui.HAnimateEffect.PLAY_REPEATING = 1;
public final static int org.havi.ui.HAnimateEffect.PLAY_ALTERNATING = 2;
public final static int org.havi.ui.HBackgroundConfigTemplate.CHANGEABLE_SINGLE_COLOR = 10;
public final static int org.havi.ui.HBackgroundConfigTemplate.STILL_IMAGE = 11;
public final static int org.havi.ui.HFontCapabilities.BASIC_LATIN = 1;
public final static int org.havi.ui.HFontCapabilities.LATIN_1_SUPPLEMENT = 2;
public final static int org.havi.ui.HFontCapabilities.LATIN_EXTENDED_A = 3;
public final static int org.havi.ui.HFontCapabilities.LATIN_EXTENDED_B = 4;
public final static int org.havi.ui.HFontCapabilities.IPA_EXTENSIONS = 5;
public final static int org.havi.ui.HFontCapabilities.SPACING_MODIFIER_LETTERS = 6;
public final static int org.havi.ui.HFontCapabilities.COMBINING_DIACRITICAL_MARKS = 7;
public final static int org.havi.ui.HFontCapabilities.BASIC_GREEK = 8;
public final static int org.havi.ui.HFontCapabilities.GREEK_SYMBOLS_AND_COPTIC = 9;
public final static int org.havi.ui.HFontCapabilities.CYRILLIC = 10;
public final static int org.havi.ui.HFontCapabilities.ARMENIAN = 11;
public final static int org.havi.ui.HFontCapabilities.BASIC_HEBREW = 12;
public final static int org.havi.ui.HFontCapabilities.HEBREW_EXTENDED = 13;
public final static int org.havi.ui.HFontCapabilities.BASIC_ARABIC = 14;
public final static int org.havi.ui.HFontCapabilities.ARABIC_EXTENDED = 15;
public final static int org.havi.ui.HFontCapabilities.DEVANAGARI = 16;
public final static int org.havi.ui.HFontCapabilities.BENGALI = 17;
public final static int org.havi.ui.HFontCapabilities.GURMUKHI = 18;
public final static int org.havi.ui.HFontCapabilities.GUJARATI = 19;
public final static int org.havi.ui.HFontCapabilities.ORIYA = 20;
public final static int org.havi.ui.HFontCapabilities.TAMIL = 21;
public final static int org.havi.ui.HFontCapabilities.TELUGU = 22;
public final static int org.havi.ui.HFontCapabilities.KANNADA = 23;
public final static int org.havi.ui.HFontCapabilities.MALAYALAM = 24;
public final static int org.havi.ui.HFontCapabilities.THAI = 25;
public final static int org.havi.ui.HFontCapabilities.LAO = 26;
public final static int org.havi.ui.HFontCapabilities.BASIC_GEORGIAN = 27;
public final static int org.havi.ui.HFontCapabilities.GEORGIAN_EXTENDED = 28;
public final static int org.havi.ui.HFontCapabilities.HANGUL_JAMO = 29;
public final static int org.havi.ui.HFontCapabilities.LATIN_EXTENDED_ADDITIONAL = 30;
public final static int org.havi.ui.HFontCapabilities.GREEK_EXTENDED = 31;
public final static int org.havi.ui.HFontCapabilities.GENERAL_PUNCTUATION = 32;
public final static int org.havi.ui.HFontCapabilities.SUPERSCRIPTS_AND_SUBSCRIPTS = 33;
public final static int org.havi.ui.HFontCapabilities.CURRENCY_SYMBOLS = 34;
public final static int org.havi.ui.HFontCapabilities.COMBINING_DIACTRICAL_MARKS_FOR_SYMBOLS = 35;
public final static int org.havi.ui.HFontCapabilities.LETTERLIKE_SYMBOLS = 36;
public final static int org.havi.ui.HFontCapabilities.NUMBER_FORMS = 37;
public final static int org.havi.ui.HFontCapabilities.ARROWS = 38;
public final static int org.havi.ui.HFontCapabilities.MATHEMATICAL_OPERATORS = 39;
public final static int org.havi.ui.HFontCapabilities.MISCELLANEOUS_TECHNICAL = 40;
public final static int org.havi.ui.HFontCapabilities.CONTROL_PICTURES = 41;
public final static int org.havi.ui.HFontCapabilities.OPTICAL_CHARACTER_RECOGNITION = 42;
public final static int org.havi.ui.HFontCapabilities.ENCLOSED_ALPHANUMERICS = 43;
public final static int org.havi.ui.HFontCapabilities.BOX_DRAWING = 44;
public final static int org.havi.ui.HFontCapabilities.BLOCK_ELEMENTS = 45;
public final static int org.havi.ui.HFontCapabilities.GEOMETRICAL_SHAPES = 46;
public final static int org.havi.ui.HFontCapabilities.MISCELLANEOUS_SYMBOLS = 47;
public final static int org.havi.ui.HFontCapabilities.DINGBATS = 48;
public final static int org.havi.ui.HFontCapabilities.CJK_SYMBOLS_AND_PUNCTUATION = 49;
public final static int org.havi.ui.HFontCapabilities.HIRAGANA = 50;
public final static int org.havi.ui.HFontCapabilities.KATAKANA = 51;
public final static int org.havi.ui.HFontCapabilities.BOPOMOFO = 52;
public final static int org.havi.ui.HFontCapabilities.HANGUL_COMPATIBILITY_JAMO = 53;
public final static int org.havi.ui.HFontCapabilities.CJK_MISCELLANEOUS = 54;
public final static int org.havi.ui.HFontCapabilities.ENCLOSED_CJK_LETTERS_AND_MONTHS = 55;
public final static int org.havi.ui.HFontCapabilities.CJK_COMPATIBILITY = 56;
public final static int org.havi.ui.HFontCapabilities.HANGUL = 57;
public final static int org.havi.ui.HFontCapabilities.HANGUL_SUPPLEMENTARY_A = 58;
public final static int org.havi.ui.HFontCapabilities.HANGUL_SUPPLEMENTARY_B = 59;
public final static int org.havi.ui.HFontCapabilities.CJK_UNIFIED_IDEOGRAPHS = 60;
public final static int org.havi.ui.HFontCapabilities.PRIVATE_USE_AREA = 61;
```

```
public final static int org.havi.ui.HFontCapabilities.CJK_COMPATIBILITY_IDEOGRAPHS = 62;
public final static int org.havi.ui.HFontCapabilities.ALPHABETIC_PRESENTATION_FORMS_A = 63;
public final static int org.havi.ui.HFontCapabilities.ARABIC_PRESENTATION_FORMS_A = 64;
public final static int org.havi.ui.HFontCapabilities.COMBINING_HALF_MARKS = 65;
public final static int org.havi.ui.HFontCapabilities.CJK_COMPATIBILITY_FORMS = 66;
public final static int org.havi.ui.HFontCapabilities.SMALL_FORM_VARIANTS = 67;
public final static int org.havi.ui.HFontCapabilities.ARABIC_PRESENTATION_FORMS_B = 68;
public final static int org.havi.ui.HFontCapabilities.HALFWIDTH_AND_FULLWIDTH_FORMS = 69;
public final static int org.havi.ui.HFontCapabilities.SPECIALS = 70;
public final static int org.havi.ui.HGraphicsConfigTemplate.VIDEO_MIXING = 12;
public final static int org.havi.ui.HGraphicsConfigTemplate.MATTE_SUPPORT = 13;
public final static int org.havi.ui.HGraphicsConfigTemplate.IMAGE_SCALING_SUPPORT = 14;
public final static int org.havi.ui.HImageHints.NATURAL_IMAGE = 1;
public final static int org.havi.ui.HImageHints.CARTOON = 2;
public final static int org.havi.ui.HImageHints.BUSINESS_GRAPHICS = 3;
public final static int org.havi.ui.HImageHints.LINE_ART = 4;
public final static int org.havi.ui.HKeyboardInputPreferred.INPUT_NUMERIC = 1;
public final static int org.havi.ui.HKeyboardInputPreferred.INPUT_ALPHA = 2;
public final static int org.havi.ui.HKeyboardInputPreferred.INPUT_ANY = 4;
public final static int org.havi.ui.HKeyboardInputPreferred.INPUT_CUSTOMIZED = 8;
public final static int org.havi.ui.HListGroup.DEFAULT_ICON_HEIGHT = -4;
public final static int org.havi.ui.HListGroup.DEFAULT_ICON_WIDTH = -3;
public final static int org.havi.ui.HListGroup.DEFAULT_LABEL_HEIGHT = -2;
public final static int org.havi.ui.HListGroup.ITEM_NOT_FOUND = -1;
public final static int org.havi.ui.HListGroup.ADD_INDEX_END = -1;
public final static int org.havi.ui.HListGroup.DEFAULT_LABEL_WIDTH = -1;
public final static int org.havi.ui.HOrientable.ORIENT_LEFT_TO_RIGHT = 0;
public final static int org.havi.ui.HOrientable.ORIENT_RIGHT_TO_LEFT = 1;
public final static int org.havi.ui.HOrientable.ORIENT_TOP_TO_BOTTOM = 2;
public final static int org.havi.ui.HOrientable.ORIENT_BOTTOM_TO_TOP = 3;
public final static int org.havi.ui.HScene.IMAGE_NONE = 0;
public final static int org.havi.ui.HScene.NO_BACKGROUND_FILL = 0;
public final static int org.havi.ui.HScene.IMAGE_STRETCH = 1;
public final static int org.havi.ui.HScene.BACKGROUND_FILL = 1;
public final static int org.havi.ui.HScene.IMAGE_CENTER = 2;
public final static int org.havi.ui.HScene.IMAGE_TILE = 3;
public final static int org.havi.ui.HSceneTemplate.GRAPHICS_CONFIGURATION = 0;
public final static int org.havi.ui.HSceneTemplate.REQUIRED = 1;
public final static int org.havi.ui.HSceneTemplate.SCENE_PIXEL_DIMENSION = 1;
public final static int org.havi.ui.HSceneTemplate.PREFERRED = 2;
public final static int org.havi.ui.HSceneTemplate.SCENE_PIXEL_LOCATION = 2;
public final static int org.havi.ui.HSceneTemplate.UNNECESSARY = 3;
public final static int org.havi.ui.HSceneTemplate.SCENE_SCREEN_DIMENSION = 4;
public final static int org.havi.ui.HSceneTemplate.SCENE_SCREEN_LOCATION = 8;
public final static int org.havi.ui.HScreenConfigTemplate.REQUIRED = 1;
public final static int org.havi.ui.HScreenConfigTemplate.ZERO_BACKGROUND_IMPACT = 1;
public final static int org.havi.ui.HScreenConfigTemplate.PREFERRED = 2;
public final static int org.havi.ui.HScreenConfigTemplate.ZERO_GRAPHICS_IMPACT = 2;
public final static int org.havi.ui.HScreenConfigTemplate.DONT_CARE = 3;
public final static int org.havi.ui.HScreenConfigTemplate.ZERO_VIDEO_IMPACT = 3;
public final static int org.havi.ui.HScreenConfigTemplate.PREFERRED_NOT = 4;
public final static int org.havi.ui.HScreenConfigTemplate.INTERLACED_DISPLAY = 4;
public final static int org.havi.ui.HScreenConfigTemplate.REQUIRED_NOT = 5;
public final static int org.havi.ui.HScreenConfigTemplate.FLICKER_FILTERING = 5;
public final static int org.havi.ui.HScreenConfigTemplate.VIDEO_GRAPHICS_PIXEL_ALIGNED = 6;
public final static int org.havi.ui.HScreenConfigTemplate.PIXEL_ASPECT_RATIO = 7;
public final static int org.havi.ui.HScreenConfigTemplate.PIXEL_RESOLUTION = 8;
public final static int org.havi.ui.HScreenConfigTemplate.SCREEN_RECTANGLE = 9;
public final static int org.havi.ui.HState.FOCUSED_STATE_BIT = 1;
public final static int org.havi.ui.HState.ACTIONED_STATE_BIT = 2;
public final static int org.havi.ui.HState.DISABLED_STATE_BIT = 4;
public final static int org.havi.ui.HState.ALL_STATES = 7;
public final static int org.havi.ui.HState.FIRST_STATE = 128;
public final static int org.havi.ui.HState.NORMAL_STATE = 128;
public final static int org.havi.ui.HState.FOCUSED_STATE = 129;
public final static int org.havi.ui.HState.ACTIONED_STATE = 130;
public final static int org.havi.ui.HState.ACTIONED_FOCUSED_STATE = 131;
```

```
public final static int org.havi.ui.HState.DISABLED_STATE = 132;
public final static int org.havi.ui.HState.DISABLED_FOCUSED_STATE = 133;
public final static int org.havi.ui.HState.DISABLED_ACTIONED_STATE = 134;
public final static int org.havi.ui.HState.DISABLED_ACTIONED_FOCUSED_STATE = 135;
public final static int org.havi.ui.HState.LAST_STATE = 135;
public final static int org.havi.ui.HStaticRange.SLIDER_BEHAVIOR = 0;
public final static int org.havi.ui.HStaticRange.SCROLLBAR_BEHAVIOR = 1;
public final static int org.havi.ui.HVideoConfigTemplate.GRAPHICS_MIXING = 15;
public final static int org.havi.ui.HVisible.NO_DEFAULT_WIDTH = -1;
public final static int org.havi.ui.HVisible.NO_DEFAULT_HEIGHT = -1;
public final static int org.havi.ui.HVisible.HALIGN_LEFT = 0;
public final static int org.havi.ui.HVisible.VALIGN_TOP = 0;
public final static int org.havi.ui.HVisible.RESIZE_NONE = 0;
public final static int org.havi.ui.HVisible.NO_BACKGROUND_FILL = 0;
public final static int org.havi.ui.HVisible.FIRST_CHANGE = 0;
public final static int org.havi.ui.HVisible.TEXT_CONTENT_CHANGE = 0;
public final static int org.havi.ui.HVisible.HALIGN_CENTER = 1;
public final static int org.havi.ui.HVisible.RESIZE_PRESERVE_ASPECT = 1;
public final static int org.havi.ui.HVisible.BACKGROUND_FILL = 1;
public final static int org.havi.ui.HVisible.GRAPHIC_CONTENT_CHANGE = 1;
public final static int org.havi.ui.HVisible.HALIGN_RIGHT = 2;
public final static int org.havi.ui.HVisible.RESIZE_ARBITRARY = 2;
public final static int org.havi.ui.HVisible.ANIMATE_CONTENT_CHANGE = 2;
public final static int org.havi.ui.HVisible.HALIGN_JUSTIFY = 3;
public final static int org.havi.ui.HVisible.CONTENT_CHANGE = 3;
public final static int org.havi.ui.HVisible.VALIGN_CENTER = 4;
public final static int org.havi.ui.HVisible.STATE_CHANGE = 4;
public final static int org.havi.ui.HVisible.CARET_POSITION_CHANGE = 5;
public final static int org.havi.ui.HVisible.ECHO_CHAR_CHANGE = 6;
public final static int org.havi.ui.HVisible.EDIT_MODE_CHANGE = 7;
public final static int org.havi.ui.HVisible.VALIGN_BOTTOM = 8;
public final static int org.havi.ui.HVisible.MIN_MAX_CHANGE = 8;
public final static int org.havi.ui.HVisible.THUMB_OFFSETS_CHANGE = 9;
public final static int org.havi.ui.HVisible.ORIENTATION_CHANGE = 10;
public final static int org.havi.ui.HVisible.TEXT_VALUE_CHANGE = 11;
public final static int org.havi.ui.HVisible.VALIGN_JUSTIFY = 12;
public final static int org.havi.ui.HVisible.ITEM_VALUE_CHANGE = 12;
public final static int org.havi.ui.HVisible.ADJUSTMENT_VALUE_CHANGE = 13;
public final static int org.havi.ui.HVisible.LIST_CONTENT_CHANGE = 14;
public final static int org.havi.ui.HVisible.LIST_ICONSIZE_CHANGE = 15;
public final static int org.havi.ui.HVisible.LIST_LABELSIZE_CHANGE = 16;
public final static int org.havi.ui.HVisible.LIST_MULTISELECTION_CHANGE = 17;
public final static int org.havi.ui.HVisible.LIST_SCROLLPOSITION_CHANGE = 18;
public final static int org.havi.ui.HVisible.SIZE_CHANGE = 19;
public final static int org.havi.ui.HVisible.BORDER_CHANGE = 20;
public final static int org.havi.ui.HVisible.REPEAT_COUNT_CHANGE = 21;
public final static int org.havi.ui.HVisible.ANIMATION_POSITION_CHANGE = 22;
public final static int org.havi.ui.HVisible.LIST_SELECTION_CHANGE = 23;
public final static int org.havi.ui.HVisible.UNKNOWN_CHANGE = 24;
public final static int org.havi.ui.HVisible.LAST_CHANGE = 24;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_FIRST = 2000;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_START_CHANGE = 2000;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_LESS = 2001;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_MORE = 2002;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_PAGE_LESS = 2003;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_PAGE_MORE = 2004;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_LAST = 2005;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_END_CHANGE = 2005;
public final static int org.havi.ui.event.HBackgroundImageEvent.BACKGROUNDIMAGE_FIRST = 1;
public final static int org.havi.ui.event.HBackgroundImageEvent.BACKGROUNDIMAGE_LOADED = 1;
public final static int org.havi.ui.event.HBackgroundImageEvent.BACKGROUNDIMAGE_FILE_NOT_FOUND
= 2;
public final static int org.havi.ui.event.HBackgroundImageEvent.BACKGROUNDIMAGE_IOERROR = 3;
public final static int org.havi.ui.event.HBackgroundImageEvent.BACKGROUNDIMAGE_INVALID = 4;
public final static int org.havi.ui.event.HBackgroundImageEvent.BACKGROUNDIMAGE_LAST = 4;
public final static int org.havi.ui.event.HEventRepresentation.ER_TYPE_NOT_SUPPORTED = 0;
public final static int org.havi.ui.event.HEventRepresentation.ER_TYPE_STRING = 1;
```

```
public final static int org.havi.ui.event.HEventRepresentation.ER_TYPE_COLOR = 2;
public final static int org.havi.ui.event.HEventRepresentation.ER_TYPE_SYMBOL = 4;
public final static int org.havi.ui.event.HFocusEvent.NO_TRANSFER_ID = -1;
public final static int org.havi.ui.event.HFocusEvent.HFOCUS_FIRST = 2029;
public final static int org.havi.ui.event.HFocusEvent.FOCUS_TRANSFER = 2029;
public final static int org.havi.ui.event.HFocusEvent.HFOCUS_LAST = 2029;
public final static int org.havi.ui.event.HItemEvent.ITEM_FIRST = 2006;
public final static int org.havi.ui.event.HItemEvent.ITEM_START_CHANGE = 2006;
public final static int org.havi.ui.event.HItemEvent.ITEM_TOGGLE_SELECTED = 2007;
public final static int org.havi.ui.event.HItemEvent.ITEM_SELECTED = 2008;
public final static int org.havi.ui.event.HItemEvent.ITEM_CLEARED = 2009;
public final static int org.havi.ui.event.HItemEvent.ITEM_SELECTION_CLEARED = 2010;
public final static int org.havi.ui.event.HItemEvent.ITEM_SET_CURRENT = 2011;
public final static int org.havi.ui.event.HItemEvent.ITEM_SET_PREVIOUS = 2012;
public final static int org.havi.ui.event.HItemEvent.ITEM_SET_NEXT = 2013;
public final static int org.havi.ui.event.HItemEvent.SCROLL_MORE = 2014;
public final static int org.havi.ui.event.HItemEvent.SCROLL_LESS = 2015;
public final static int org.havi.ui.event.HItemEvent.SCROLL_PAGE_MORE = 2016;
public final static int org.havi.ui.event.HItemEvent.SCROLL_PAGE_LESS = 2017;
public final static int org.havi.ui.event.HItemEvent.ITEM_END_CHANGE = 2018;
public final static int org.havi.ui.event.HItemEvent.ITEM_LAST = 2018;
public final static int org.havi.ui.event.HRcEvent.RC_FIRST = 400;
public final static int org.havi.ui.event.HRcEvent.VK_COLORED_KEY_0 = 403;
public final static int org.havi.ui.event.HRcEvent.VK_COLORED_KEY_1 = 404;
public final static int org.havi.ui.event.HRcEvent.VK_COLORED_KEY_2 = 405;
public final static int org.havi.ui.event.HRcEvent.VK_COLORED_KEY_3 = 406;
public final static int org.havi.ui.event.HRcEvent.VK_COLORED_KEY_4 = 407;
public final static int org.havi.ui.event.HRcEvent.VK_COLORED_KEY_5 = 408;
public final static int org.havi.ui.event.HRcEvent.VK_POWER = 409;
public final static int org.havi.ui.event.HRcEvent.VK_DIMMER = 410;
public final static int org.havi.ui.event.HRcEvent.VK_WINK = 411;
public final static int org.havi.ui.event.HRcEvent.VK_REWIND = 412;
public final static int org.havi.ui.event.HRcEvent.VK_STOP = 413;
public final static int org.havi.ui.event.HRcEvent.VK_EJECT_TOGGLE = 414;
public final static int org.havi.ui.event.HRcEvent.VK_PLAY = 415;
public final static int org.havi.ui.event.HRcEvent.VK_RECORD = 416;
public final static int org.havi.ui.event.HRcEvent.VK_FAST_FWD = 417;
public final static int org.havi.ui.event.HRcEvent.VK_PLAY_SPEED_UP = 418;
public final static int org.havi.ui.event.HRcEvent.VK_PLAY_SPEED_DOWN = 419;
public final static int org.havi.ui.event.HRcEvent.VK_PLAY_SPEED_RESET = 420;
public final static int org.havi.ui.event.HRcEvent.VK_RECORD_SPEED_NEXT = 421;
public final static int org.havi.ui.event.HRcEvent.VK_GO_TO_START = 422;
public final static int org.havi.ui.event.HRcEvent.VK_GO_TO_END = 423;
public final static int org.havi.ui.event.HRcEvent.VK_TRACK_PREV = 424;
public final static int org.havi.ui.event.HRcEvent.VK_TRACK_NEXT = 425;
public final static int org.havi.ui.event.HRcEvent.VK_RANDOM_TOGGLE = 426;
public final static int org.havi.ui.event.HRcEvent.VK_CHANNEL_UP = 427;
public final static int org.havi.ui.event.HRcEvent.VK_CHANNEL_DOWN = 428;
public final static int org.havi.ui.event.HRcEvent.VK_STORE_FAVORITE_0 = 429;
public final static int org.havi.ui.event.HRcEvent.VK_STORE_FAVORITE_1 = 430;
public final static int org.havi.ui.event.HRcEvent.VK_STORE_FAVORITE_2 = 431;
public final static int org.havi.ui.event.HRcEvent.VK_STORE_FAVORITE_3 = 432;
public final static int org.havi.ui.event.HRcEvent.VK_RECALL_FAVORITE_0 = 433;
public final static int org.havi.ui.event.HRcEvent.VK_RECALL_FAVORITE_1 = 434;
public final static int org.havi.ui.event.HRcEvent.VK_RECALL_FAVORITE_2 = 435;
public final static int org.havi.ui.event.HRcEvent.VK_RECALL_FAVORITE_3 = 436;
public final static int org.havi.ui.event.HRcEvent.VK_CLEAR_FAVORITE_0 = 437;
public final static int org.havi.ui.event.HRcEvent.VK_CLEAR_FAVORITE_1 = 438;
public final static int org.havi.ui.event.HRcEvent.VK_CLEAR_FAVORITE_2 = 439;
public final static int org.havi.ui.event.HRcEvent.VK_CLEAR_FAVORITE_3 = 440;
public final static int org.havi.ui.event.HRcEvent.VK_SCAN_CHANNELS_TOGGLE = 441;
public final static int org.havi.ui.event.HRcEvent.VK_PINP_TOGGLE = 442;
public final static int org.havi.ui.event.HRcEvent.VK_SPLIT_SCREEN_TOGGLE = 443;
public final static int org.havi.ui.event.HRcEvent.VK_DISPLAY_SWAP = 444;
public final static int org.havi.ui.event.HRcEvent.VK_SCREEN_MODE_NEXT = 445;
public final static int org.havi.ui.event.HRcEvent.VK_VIDEO_MODE_NEXT = 446;
public final static int org.havi.ui.event.HRcEvent.VK_VOLUME_UP = 447;
```

```
public final static int org.havi.ui.event.HRcEvent.VK_VOLUME_DOWN = 448;
public final static int org.havi.ui.event.HRcEvent.VK_MUTE = 449;
public final static int org.havi.ui.event.HRcEvent.VK_SURROUND_MODE_NEXT = 450;
public final static int org.havi.ui.event.HRcEvent.VK_BALANCE_RIGHT = 451;
public final static int org.havi.ui.event.HRcEvent.VK_BALANCE_LEFT = 452;
public final static int org.havi.ui.event.HRcEvent.VK_FADER_FRONT = 453;
public final static int org.havi.ui.event.HRcEvent.VK_FADER_REAR = 454;
public final static int org.havi.ui.event.HRcEvent.VK_BASS_BOOST_UP = 455;
public final static int org.havi.ui.event.HRcEvent.VK_BASS_BOOST_DOWN = 456;
public final static int org.havi.ui.event.HRcEvent.VK_INFO = 457;
public final static int org.havi.ui.event.HRcEvent.VK_GUIDE = 458;
public final static int org.havi.ui.event.HRcEvent.VK_TELETEXT = 459;
public final static int org.havi.ui.event.HRcEvent.VK_SUBTITLE = 460;
public final static int org.havi.ui.event.HRcEvent.RC_LAST = 460;
public final static int org.havi.ui.event.HTextEvent.TEXT_FIRST = 2019;
public final static int org.havi.ui.event.HTextEvent.TEXT_START_CHANGE = 2019;
public final static int org.havi.ui.event.HTextEvent.TEXT_CHANGE = 2020;
public final static int org.havi.ui.event.HTextEvent.TEXT_CARET_CHANGE = 2021;
public final static int org.havi.ui.event.HTextEvent.TEXT_END_CHANGE = 2022;
public final static int org.havi.ui.event.HTextEvent.CARET_NEXT_CHAR = 2023;
public final static int org.havi.ui.event.HTextEvent.CARET_NEXT_LINE = 2024;
public final static int org.havi.ui.event.HTextEvent.CARET_PREV_CHAR = 2025;
public final static int org.havi.ui.event.HTextEvent.CARET_PREV_LINE = 2026;
public final static int org.havi.ui.event.HTextEvent.CARET_NEXT_PAGE = 2027;
public final static int org.havi.ui.event.HTextEvent.TEXT_LAST = 2028;
public final static int org.havi.ui.event.HTextEvent.CARET_PREV_PAGE = 2028;
```

9 SDD Data

9.1 References

At the beginning of this document is a collection of references. Each of the IEEE 1212 ROM fields defined in this section includes the reference indicator, either [1], [2] or [4] for convenience to the reader. Fields which are defined in this document specifically for the HAVi architecture will be appended with [HAVi] as the reference indicator.

9.2 Introduction

This section describes the SDD data for HAVi-compliant 1394 devices. This information is stored in the configuration ROM of the device, according to the layout rules defined by [1] and [2].

About the IEEE 1212 specification: the current official draft is [1]. However, the specification is currently undergoing a revision process as part of its 5-year re-evaluation. Some of the newly defined items for this revision are necessary for the HAVi architecture. The newly defined data structures are in [2]. Additional HAVi-specific data structures are defined in this document.

9.3 Text Encoding Formats

1212 reference [1] defines only a minimal English ASCII text encoding format. The 1212 revision [2] defines several additional 2-byte formats, including UNICODE. The text fields of SDD use UNICODE encoding.

9.4 HAVi Key Values

Each of the keys in the following table is defined within the scope of the HAVi Unit Directory. The definitions for each of these keys are presented in subsequent sections of this document.

Table 17.   HAVi Unit Directory Key Values

HAVi Key Name	Key ID	Valid Key Types
HAVi_Device_Profile_Key	38_{16}	Immediate
HAVi_DCM_Key	39_{16}	Leaf offset
HAVi_DCM_Reference_Key	$3A_{16}$	Leaf offset
HAVi_DCM_Profile_Key	$3B_{16}$	Leaf offset
HAVi_Device_Icon_Bitmap_Key	$3C_{16}$	Leaf offset
HAVi_Message_Version_Key	$3D_{16}$	Immediate

9.5 Minimum Required Data

The fields described in this document are the minimum required for HAVi-compliant devices, but these may not be all that is necessary for the device being implemented. Depending on other standards supported by the device, additional fields may be required.

If a device changes any of the HAVi defined fields, except the user preferred name, it shall generate a 1394 bus reset. Moreover, a HAVi controller must also generate a bus reset after initialization or self-test, i.e., when the first quadlet of its configuration ROM transits from a zero to non-zero value. This bus reset is necessary to re-detect HAVi controller devices and restart HAVi specific protocols between controllers.

The following table illustrates the required and optional HAVi-specific data:

Table 18.   HAVi Configuration ROM Requirements

HAVi Key Name	Requirement
HAVi_Device_Profile_Key	Required for all HAVi-compliant device classes
modifiable descriptor for HAVi_Device_Profile	Required for all HAVi-compliant device classes
HAVi_DCM_Key	Strongly recommended for BAV Does not apply for IAV, FAV
HAVi_DCM_Reference_Key	Strongly recommended for BAV Does not apply for IAV, FAV
HAVi_DCM_Profile_Key	Required if DCM is included (BAV) Does not apply for IAV, FAV
HAVi_Device_Icon_Bitmap_Key	Optional for all HAVi-compliant device classes
HAVi_Message_Version_Key	Required for IAV, FAV, does not apply for BAV

The following table illustrates the required non-HAVi-specific data (defined by other standards, such as [1] and [2]):

Table 19.   Non-HAVi Configuration ROM Requirements

Key_ID	Key Name	Requirement	Directory	Described in [2]
18_{16}	Instance_Directory	Required for all HAVi device classes	Root directory	7.7.15
17_{16}	Model_ID	Required for all HAVi device classes	Root directory (exceptionally vendor or instance directory)	7.7.14
11_{16}	HAVi_Unit_Directory	Required for all HAVi device classes	Instance directory (and possibly root)	7.7.9
11_{16}	IEC_61883_Unit_Directory	Required for HAVi devices supporting multiple 61883 protocols	Root and instance directory	7.7.9
12_{16}	Specifier_ID	Required for all HAVi device classes	HAVi unit directory	7.7.10
13_{16}	Version	Required for all HAVi device classes	HAVi unit directory	7.7.11

9.6 ROM Format

Devices shall implement the general ROM format as defined in [1].

9.7 The GUID and the Bus_Info_Block

In the Bus_Info_Block of the configuration ROM, the 8 bit chip_id_hi field is concatenated with the 32 bit chip_id_lo to create a 40 bit chip ID value. The vendor specified by the node_vendor_id value shall administer the chip ID value. When appended to the node_vendor_id value, these shall form a unique 64-bit value called EUI-64 (Extended Unique Identifier, 64 bits). The HAVi GUID (Global Unique Identifier) reflects that EUI-64 value.

9.8 Root Directory

Devices shall implement a root directory with the following fields:

9.8.1 Vendor_ID [2]

The Vendor_ID is a 24-bit immediate value (registration ID), assigned by the IEEE, which is globally unique. This value identifies the vendor that manufactured the device.

9.8.2 HAVi_Unit_Directory [1]

1394 devices conformant with the HAVi specification are required to support the IEC 61883 standard for data transmission formats. All devices shall implement a Unit_Directory field which points to a 1212 Unit Directory which specifies the IEC 61883 protocol and the HAVi protocol. This structure is described in more detail below.

The HAVi_Unit_Directory entry is mandatory in the instance directory. A second entry in the root directory is present only to allow older devices compatible with the current version of the IEC 61883 standard to find this unit directory. If present, the HAVi_Unit_Directory entry shall be the only unit directory entry in the root, i.e., the HAVi_Unit_Directory entry will be present in the root only if no other IEC_61883_Unit_Directory (see section 9.8.3) is present in the root. Since the p1212 revision [2] obsoletes this entry, HAVi devices shall rely only on the HAVi_Unit_Directory entry of the instance directory.

9.8.3 Other IEC_61883_Unit_Directory [1] [4]

HAVi devices may optionally implement further protocols specified by the IEC 61883 CTS code. If this is the case, a separate unit directory is needed for those non-HAVi protocols.

9.8.4 Instance_Directory [2]

The newly defined Instance_Directory is an offset reference to a directory structure. An instance directory is mandatory for every HAVi device, since it contains the link for the HAVi Unit Directory.

9.8.5 Model_ID [2]

The Model_ID field is a binary value which identifies the device model. The format of this field is 24-bit immediate. The contents of the Model_ID are defined by the device vendor. The Model_ID value should represent a family or class of products and should not be unique to individual devices.

HAVi devices shall, if possible, have the Model_ID entry in the root directory. If this is not possible,

e.g., for multi-standard devices, the Model_ID may exceptionally be present in a root dependent vendor directory or the instance directory that contains the HAVi Unit Directory. If a Model_ID is present in both (and not in the root directory), the entries (and dependent descriptors) shall be identical. If a Model_ID entry is present in the root and elsewhere, the entries (and dependent descriptors) should be identical as HAVi considers only the root directory.

9.9 Instance Directory

Devices shall implement an Instance_Directory with the following fields:

9.9.1 HAVi_Unit_Directory [1][2]

This entry points to the same HAVi_Unit_Directory (see below). A second entry pointing to the HAVi_Unit_Directory may under certain conditions be present in the root directory. According to [2] HAVi devices shall rely only on the HAVi_Unit_Directory entry in the instance directory, the entry in the root directory is needed only for backwards compatibility.

9.10 HAVi Unit Directory

Devices shall declare a unit architecture with the following fields as a minimum:

9.10.1 Specifier_ID [1]

The Specifier_ID shall be the first entry of the Unit Directory.

The Specifier_ID in the HAVi Unit Directory is a 24-bit immediate value, and shall be that of the 1394 Trade Association, as follows:

first octet	00_{16}
second octet	$A0_{16}$
third octet	$2D_{16}$

9.10.2 Version [1]

The Version entry shall be the second entry in the Unit Directory.

In the Version field the two least significant bytes specify the version of the SDD fields which are defined in this device, as described below:

first octet	01_{16}
second octet	00_{16}
third octet	08_{16}

The combination of Specifier_ID and Version defines the meaning of all subsequent keys in the range of 38_{16} to $3F_{16}$ inclusive.

9.10.3 HAVi_Message_Version [HAVi]

The HAVi_Message_Version is a 24-bit immediate value, which specifies the version of the HAVi Messaging System supported by this device. It has the following definition:

first octet	Reserved (currently 00_{16})
second octet	Major Version Number (currently 01_{16})
third octet	Minor Version Number (currently $0A_{16}$)

9.10.4   HAVi_Device_Profile [HAVi]

The HAVi_Device_Profile is a 24-bit immediate value specifying the main capabilities of the device. The value is interpreted as a field of 24 bits with individual meanings, in the following description bits are numbered from Bit 0 for the LSB to Bit 23 for the MSB.

9.10.4.1   HAVi_Device_Class [Bit0..3]

The HAVi_Device_Class is a 4-bit immediate value specifying which of the non-LAV device categories this device conforms to. The following values are defined:

HAVi_Device_Class value	Meaning
0000_2	reserved
0001_2	BAV Device
0010_2	IAV Device
0011_2	FAV Device
other values	reserved

9.10.4.2   HAVi_DCM_Manager [Bit4]

The HAVi_DCM_Manager is a 1-bit immediate value specifying for IAV devices whether a DCM Manager is implemented. For a BAV this bit shall be 0, for a FAV this bit shall be 1.

HAVi_DCM_Manager value	Meaning
0	DCM Manager absent
1	DCM Manager present

9.10.4.3   HAVi_Stream_Manager [Bit5]

The HAVi_Stream_Manager is a 1-bit immediate value specifying for IAV devices whether a Stream Manager is implemented. For a BAV this bit shall be 0, for a FAV this bit shall be 1.

HAVi_Stream_Manager value	Meaning
0	Stream Manager absent
1	Stream Manager present

9.10.4.4   HAVi_Resource_Manager [Bit6]

The HAVi_Resource_Manager is a 1-bit immediate value specifying for IAV devices whether a Resource Manager is implemented. For a BAV this bit shall be 0, for a FAV this bit shall be 1.

HAVi_Resource_Manager value	Meaning
0	Resource Manager absent

1	Resource Manager present

9.10.4.5   HAVi_Display_Capability [Bit7]

The HAVi_Display_Capability is a 1-bit immediate value specifying for IAV devices whether a DDI Controller is implemented, and for FAV devices whether a DDI Controller and a Level 2 User Interface capability is implemented. For a BAV this bit shall be 0.

HAVi_Display_Capability value	Meaning
0	does not have display capability
1	has display capability

9.10.4.6   HAVi_Device_Status [Bit8]

This bit specifies the status of the device. For an IAV or FAV, a value of one indicates that the HAVi Messaging System and other system components are running – the device is prepared to receive and process incoming HAVi messages. For a BAV, a value of one indicates that the device is ready to accept native commands (e.g., AV/C) over 1394.

HAVi_Device_Status value	Meaning
0	inactive
1	active

9.10.4.7   Reserved Bits [Bit9..23]

The upper 15 bits of the HAVi_Device_Profile are reserved and shall read as 0.

9.10.5   HAVi_User_Preferred_Name [2][HAVi]

The HAVi_User_Preferred_Name entry is mandatory for all devices. It is coded as a single modifiable textual descriptor to the HAVi_Device_Profile. The device shall allow this textual descriptor to be modified and should store it persistently. When it is modified, the DCM must update the HAVi Registry. The means by which the device informs the DCM of an update (polling, notification ...) is implementation dependent.

From HAVi devices, this field is modified by Dcm::SetUserPreferredName. The DCM then writes the user preferred name into the node's address space, while respecting the rules for modifiable descriptors. Non-HAVi controllers shall directly modify the user preferred name in the node's address space, while respecting the rules for modifiable descriptors. Both HAVi and non-HAVi controllers shall ensure that Width and Character Set are always set to fixed two byte unicode characters, the language field shall be set to all 0, to indicate "undefined". Note that the parameter leaf referenced by the HAVi_User_Preferred_Name entry contains a fixed max_descriptor_size value of 11, to allow values of up to 16 2-byte Unicode characters.

In multi-standard devices, nicknames in addition to the HAVi_User_Preferred_Name may exist (e.g., textual descriptors to the Model_ID). HAVi-aware controllers may in this case update all names in a coherent manner, using the same text with the appropriate character set. But other controllers may not be HAVi aware, and may update the "foreign" nickname without modifying the HAVi_User_Preferred_Name. Furthermore, some HAVi controllers may update only the HAVi_User_Preferred_Name. To avoid inconsistency it is expected that in such cases the device itself attempts updating its unmodified names.

9.10.6 HAVi_DCM [HAVi]

The HAVi_DCM is a 24-bit offset to a 1212 leaf structure. The leaf contains the DCM bytecode unit. DCM code units shall be encoded as described in section 7.4.1. This field and the leaf structure are not applicable for the IAV and FAV device classes; they are recommended for the BAV device class.

9.10.7 HAVi_DCM_Profile [HAVi]

The HAVi_DCM_Profile is a leaf structure which contains information about the DCM. It is structured as follows (the following diagram represents a quadlet-aligned structure with the least significant bytes on the right):

Figure 44. HAVi_DCM_Profile Leaf Structure

Leaf length		CRC	
Transferred_DCM_Code_Unit_Size			
Installed_DCM_Code_Space			
Installed_DCM_Working_Space			
reserved		Message_Version	

The Transferred_DCM_Code_Unit_Size field specifies the size, in bytes, of the DCM code unit as it would be transferred from source to destination (i.e., JAR file size). The Installed_DCM_Code_Space designates the required memory size of the installed code unit part (read-only), not including the amount of working space the code unit requires. The Installed_DCM_Working_Space is an estimate of the working space (read/write) the code unit requires. Note that an installed DCM code unit includes installed DCM and FCMs embedded in the code unit.

The Message_Version field specifies the lowest version of the HAVi Messaging System required by this DCM. This field is interpreted as defined in the HAVi Messaging System description.

9.10.8 HAVi_DCM_Reference [HAVi]

The HAVi_DCM_Reference is a URL which provides remote access to a BAV DCM code unit and its profile. This field is implemented as a leaf structure. It is a free-standing text string which contains the URL. The HAVi_DCM_Reference field and the associated URL leaf structure are not applicable for the IAV and FAV device classes; they are recommended for the BAV device class.

For more details on the use of the URL, please refer to the DCM Manager chapter of this document.

The HAVi_DCM_Reference leaf offset points to a 1212 leaf structure which contains the URL for a remotely stored DCM code unit and its profile. It shall not contain the file name extension of either bytecode unit or profile. It is encoded as follows:

Figure 45. HAVi_DCM_Reference Leaf Structure

leaf length		CRC		
begin URL data				
	end URL data	pad bytes	if necessary	

The HAVi_DCM_Reference is simply a stream of ASCII text bytes that describe a complete URL. The string shall be NULL-terminated. If the last quadlet is not completely filled with URL data, then null pad bytes shall be added to the end, to fill the last quadlet.

Here's an example of the web address `http://www.mycompany.com/pub/misc/digitalcam` stored in the leaf structure:

Figure 46. Example of HAVi_DCM_Reference Leaf Structure

leaf length		CRC		
"h"	"t"	"t"	"p"	
":"	"/"	"/"	"w"	
"w"	"w"	"."	"m"	
"y"	"c"	"o"	etc.	

Note – the format of the leaf containing a URL is not a textual descriptor, and therefore does not have to be formatted as such. The simple embedding of ASCII characters for the URL is sufficient.

9.10.9 HAVi_Device_Icon_Bitmap [HAVi]

The HAVi_Device_Icon_Bitmap is a 24-bit offset address which points to a leaf; the leaf contains the bitmap data. The format of the bitmap encoding is described in section 5.12.5.2.

9.11 Examples (Informative)

The following examples illustrate the concepts of defining 1212 ROM structures for HAVi devices.

9.11.1 Using Keys in the Range of 38_{16} to $3F_{16}$

Data fields in the 1212 ROM are segregated by keys, which are indicators of the data which follows. The {key, data} pair allows a controller to parse a 1212 ROM and skip around those fields which it does not understand.

While most of the 6 bit key_IDs are allocated for definition by the IEEE 1212 standard, the meaning of the last eight keys ($38_{16}...3F_{16}$) is defined by the organization or vendor identified by the directory's Specifier_ID entry. The meaning of these keys depends on the Specifier_ID and the Version entry present in the same directory as the keys.

The HAVi keys described in the present specification are defined for exclusive use inside the HAVi Unit Directory. This directory contains as Specifier_ID that of the 1394 Trade Association (00 A0 2D) and in the Version field the value 01 00 08, identifying the HAVi specification as the one ruling

the use of all the keys in the range of 38_{16} to $3F_{16}$.

For example, HAVi defines key 39_{16} to mean **HAVi_DCM_Key**. Also, Company X might define key 39_{16} to mean private_device_information pointer. When a controller is scanning the ROM, if it sees the HAVi **Specifier_ID** (1394_ta_spec_id), the **Version** (01 00 08 for HAVi) and then key 39_{16}, it understands the meaning as **HAVi_DCM_Key**. If anywhere else in the ROM hierarchy, the controller finds the key 39_{16}, the controller will either understand the key's meaning or it will know that it does not understand the key's meaning and so can ignore the key.

Note that the revised 1212 also specifies extended keys, of 24-bit length, but these are not used in the present version of the HAVi specification.

9.11.2 HAVi 1212 ROM Encoding

The following diagram illustrates a HAVi 1212 ROM, using the fields defined in this document. Note that there may be several other structures required, based on other protocols to which the device conforms. This diagram contains information for a device which also complies to the IEC 61883 protocol.

9.11.2.1 Bus_Info_Block and Root Directory

The ROM header (**Bus_Info_Block**) and root directory appear as follows:

offset   (Base Address FFFF F000 0000_{16})

Bus_Info_Block

offset														
04 00_{16}	04_{16}						crc_length			rom_crc_value				
04 04_{16}	"1394"													
04 08_{16}	I r m c	c m c	i s c	b m c	p m c	r (3)	cyc_clk_acc (8)	max_rec (4)	r (2)	m ax _r o m (2)	Generation (4)	r	link_ spd (3)	
04 $0C_{16}$	node_vendor_id (24)									chip_id_hi (8)				
04 10_{16}	chip_id_lo (32)													

Root_Directory

offset		
04 14_{16}	root_length	CRC
04 18_{16}	03_{16}	model_vendor_id
04 $1C_{16}$	$0C_{16}$	node_capabilities
	17_{16}	Model_ID
	$D8_{16}$	instance directory offset
	$D1_{16}$	unit directory offset (HAVi or other IEC 61883) (for 61883 compatibility only)
	<< possibly other manufacturer-specific definitions here >>	
	. . . . .	

9.11.2.2 Instance_Directory

The **Instance_Directory** portion of the ROM is referenced by the instance directory offset field in the root directory. It appears as follows.

Figure 47. Instance_Directory (Root Dependent Directory)

directory length		CRC
$D1_{16}$	unit directory offset (HAVi)	
$D1_{16}$	unit directory offset (other IEC 61883)	
.....	<< possibly other fields >>	
.....	......	

9.11.2.3   HAVi_Unit_Directory

The HAVi Unit Directory, referenced by the field labeled "unit directory offset (HAVi)" in the instance directory appears as follows:

Figure 48. HAVi_Unit_Directory (Instance Dependent Directory)

directory length		CRC
12_{16}	Unit_Spec_ID (1394 TA = 00 A0 $2D_{16}$)	
13_{16}	Unit_SW_Version (= 01 00 08_{16})	
$3D_{16}$	HAVi_Message_Version immediate value (= 00 01 01_{16})	
38_{16}	HAVi_Device_Profile value	
$9F_{16}$	offset to modifiable descriptor parameter leaf	
$B9_{16}$	HAVi_DCM leaf offset	
BB_{16}	HAVi_DCM_Profile leaf offset	
BA_{16}	HAVi_DCM_Reference leaf offset	
BC_{16}	HAVi_Device_Icon_Bitmap leaf offset	
.....	<< possibly other fields >>	
.....	....	

9.11.2.4   Other IEC_61883_Unit_Directory

Figure 49. Unit_Directory (IEC 61883) Root Dependent Directory

directory length		CRC
12_{16}	Unit_Spec_ID (1394 TA = 00 A0 $2D_{16}$)	
13_{16}	Unit_SW_Version (= 01 XX XX_{16})	
.....	<< possibly other fields >>	
.....	....	

In the Unit_SW_Version field the least significant bytes specify any non-HAVi CTS codes specified in [4].

9.11.2.5 Modifiable Descriptor Entries for User Preferred Name

Figure 50. Descriptor Parameter Leaf

leaf length (=2)	CRC
max_name_size (11)	descriptor_address_hi
descriptor_address_lo	

Figure 51. User Preferred Name Leaf in a Modifiable Region of Configuration ROM

leaf length (4)		CRC	
descriptor_type(0)	specifier_ID (0)		
width(1)	char_set (1000)	language (0)	
0041_{16} "A"		0062_{16} "b"	
0043_{16} "C"		0000_{16}	

This example shows the name "AbC", written with 2-byte UNICODE characters.

464